(C) Evidence for Claims 1-27 – Entered by Examiner

The following items (1) – (33) listed below are hereby entered as evidence entered by the Examiner. Also listed for each item is where said evidence was entered into the record by the Examiner.

(1) Copy of US Patent Number 6631417 ("Balabine"). This evidence was entered into the record by the Examiner on page 2 paragraph 2 of the Office Action mailed 10/06/2004, on IDS sheet included in the Office Action mailed 10/06/2004.

(2) Copy of US Patent Number 6385643 ("Jacobs et al."). This evidence was entered into the record by the Examiner on page 2 paragraph 2 of the Office Action mailed 10/06/2004, on IDS sheet included in the Office Action mailed 10/06/2004.

(3) Copy of US Patent Number 6751671 ("Urien"). This evidence was entered into the record by the Examiner on page 2 paragraph 2 of the Office Action mailed 01/31/2005, on IDS sheet included in the Office Action mailed 01/31/2005.

(4) Copy of US Patent Number 6795851 ("Noy"). This evidence was entered into the record by the Examiner on page 2 paragraph 2 of the Office Action mailed 08/12/2005, and on IDS sheet included in the Office Action mailed 08/12/2005

(5) Copy of US Patent Number 6236999 ("Jacobs et al."). This evidence was entered into the record by the Examiner on page 2 paragraph 2 of the Office Action mailed 08/12/2005, and on IDS sheet included in the Office Action mailed 08/12/2005.

(6) Copy of US Patent Application Number 20020083183 ("Pujare et al."). This evidence was entered into the record by the Examiner on page 4 paragraph 5 of the Office Action mailed 08/12/2005, and on IDS sheet included in the Office Action mailed 08/12/2005.

(7) Copy of US Patent Application Number 20020083183 ("Pujare et al."). This evidence was entered into the record by the Examiner on page 4 paragraph 5 of the Office Action mailed 08/12/2005, and on IDS sheet included in the Office Action mailed 08/12/2005.

(8) Copy of US Patent Application Number 20030009571 ("Bavadekar"). This evidence was entered into the record by the Examiner on page 4 paragraph 3 of the Office Action mailed 03/23/2006, on IDS sheet included in the Office Action mailed 03/23/2006, and on IDS sheet included in the Office Action mailed 08/12/2005.

(9) Copy of US Patent Number 6112246 ("Horbal et al."). This evidence was entered into the record by the Examiner on Notice of References Cited line C of the Office Action mailed 10/06/2004.

(10) Copy of US Patent Number 6581088 ("Jacobs et al."). This evidence was entered into the record by the Examiner on Notice of References Cited line D of the Office Action mailed 10/06/2004.

(11) Copy of US Patent Number 5999979 ("Vallanki et al."). This evidence was entered into the record by the Examiner on Notice of References Cited line A of the Office Action mailed 01/31/2005.

(12) Copy of US Patent Number 5870549 ("Bobo, II"). This evidence was entered into the record by the Examiner on page 4 paragraph 13 of the Office Action mailed 01/31/2005, on IDS sheet included in the Office Action mailed 01/31/2005.

(13) Copy of US Patent Number 6658463 ("Dillon et al."). This evidence was entered into the record by the Examiner on page 4 paragraph 13 of the Office Action mailed 01/31/2005, on IDS sheet included in the Office Action mailed 01/31/2005.

(14) Copy of US Patent Number 6128653 ("del Val et al."). This evidence was entered into the record by the Examiner on Notice of References Cited line E of the Office Action mailed 01/31/2005.

(15) Copy of US Patent Number 6795848 ("Border et al."). This evidence was entered into the record by the Examiner on Notice of References Cited line F of the Office Action mailed 01/31/2005.

(16) Copy of US Patent Application Number 200291763 ("Shah et al."). This evidence was entered into the record by the Examiner on Notice of References Cited line D of the Office Action mailed 08/12/2005.

(17) Copy of US Patent Number 6412009 ("Erickson et al."). This evidence was entered into the record by the Examiner on Notice of References Cited line E of the Office Action mailed 08/12/2005.

(18) Copy of US Patent Application Number 20030009571 ("Bavadekar"). This evidence was entered into the record by the Examiner on page 4 paragraph 3 of the Office Action mailed 03/23/2006, on IDS sheet included in the Office Action mailed 03/23/2006, and on IDS sheet included in the Office Action mailed 08/12/2005.

(19) Copy of US Patent Application Number 2002161904 ("Tredoux et al."). This evidence was entered into the record by the Examiner on Notice of References Cited line G of the Office Action mailed 08/12/2005.

(20) Copy of US Patent Application Number 2003182431 ("Sturniolo et al."). This evidence was entered into the record by the Examiner on Notice of References Cited line B of the Office Action mailed 03/23/2006.

(21) Copy of US Patent Number 6941377 ("Diamant et al."). This evidence was entered into the record by the Examiner on Notice of References Cited line C of the Office Action mailed 03/23/2006.

(22) Copy of NPL - Layer 4+ Switching with QOS Support for RTP and HTTP; Harbaum, T.; IEEE 1999. This evidence was entered into the record by the Examiner on Notice of References Cited line U of the Office Action mailed 10/06/2004.

(23) Copy of NPL - Optimizing TCP Forwarder Performance; Spatscheck, O; IEEE/ACM transactions on Networking, Vol. 8, No. 2, APRIL 2000. This evidence was entered into the record by the Examiner on Notice of References Cited line V of the Office Action mailed 10/06/2004.

(24) Copy of NPL - Kernal Mechanisms for Service Differentiation in Overloaded Web Servers; Voigt, Tewari, Freimuth (2001); www.sics.se/~thiemo/usenix01.ps. This evidence was entered into the record by the Examiner on Notice of References Cited line W of the Office Action mailed 10/06/2004.

(25) Copy of NPL - TRIAD: A New Next-Generation Internet Architecture - Cheriton, Gritter (2000); www-dsg.stanford.edu/triad/triad.ps.gz. This evidence was entered into the record by the Examiner on Notice of References Cited line X of the Office Action mailed 10/06/2004.

(26) Copy of NPL - Distributed Packet Rewriting and its Application to Scalable Server Architectures; Bestavros, A; Crovella, M.; Liu, J; Martin, D.; (Oct 1998); www.cs.bu.edu/faculty/best/res/papers/icnp98.ps. This evidence was entered into the record by the Examiner on Notice of References Cited line U of the Office Action mailed 01/31/2005.

(27) Copy of NPL - Memex: A browsing assistant for collaborative archiving and mining of surf trails; Chakrabarti, S; Srivastava, S; Subramanyam, M; Tiwari, M; (2000); www.vidb.org/conf/2000/P603.pdf. This evidence was entered into the record by the Examiner on Notice of References Cited line V of the Office Action mailed 01/31/2005.

(28) Copy of NPL - APNNed goes Internet; Kemper, P., Tepper, C.; IS4-www.infomatik.uni-dortmund.de/QM/MA/pk/publication_ps_files/awpn01.ps.gz. This evidence was entered into the record by the Examiner on Notice of References Cited line W of the Office Action mailed 01/31/2005.

(29) Copy of NPL - Lightweight, Dynamic and Programmable Virtual Private Networks; Isaacs, R; (2000) www.cl.cam.ac.uk/Research/SRG/netos/ncam/docs/public/openarch00.ps.gz. This evidence was entered into the record by the Examiner on Notice of References Cited line U of the Office Action mailed 08/12/2005.

(30) Copy of NPL - Experiences from extending a legacy system with CORBA components; COT/4-08-V1.3; www.cit.dk/COT/reports/reports/Case4/08/cot-4-08.pdf. This evidence was entered into the record by the Examiner on Notice of References Cited line V of the Office Action mailed 08/12/2005.

(31) Copy of NPL - Transparent Caching of Web Services for Mobile Devices; Elbashir, K; Multi-Agent bistrica.uask.ca/madmuc/Pubs/kamal880.pdf. This evidence was entered into the record by the Examiner on Notice of References Cited line W of the Office Action mailed 08/12/2005.

(32) Copy of NPL - How to Turn a GSM SIM into a Web Server - Projecting Mobile Trust onto the World Wide Web; Guthery; Kehr; Posegga; www.teco.edu/~posegga/papers/WebSIM_Cards_Submission.pdf. This evidence was

entered into the record by the Examiner on Notice of References Cited line V of the Office Action mailed 03/23/2006.

(33) Copy of NPL - A Database Computer Attacks for the Evaluation of Intrusion Detection Systems; Kendall, K. (1999) www.kkendall.org/files/thesis/krkthesis.pdf. This evidence was entered into the record by the Examiner on Notice of References Cited line U of the Office Action mailed 03/23/2006.

Copies of all References follows.

//

US006112246A

United States Patent [19]

Horbal et al.

[11] **Patent Number: 6,112,246**

[45] **Date of Patent: Aug. 29, 2000**

[54] **SYSTEM AND METHOD FOR ACCESSING INFORMATION FROM A REMOTE DEVICE AND PROVIDING THE INFORMATION TO A CLIENT WORKSTATION**

[76] Inventors: **Mark T. Horbal**, 32802 Fowler Cir., Warrenville, Ill. 60555; **Randal J. King**, 3 S. 947 Thornapple Tree Rd., Sugar Grove, Ill. 60554

[21] Appl. No.: **09/176,993**

[22] Filed: **Oct. 22, 1998**

[51] **Int. Cl.**^{7} **G08C 15/06**

[52] **U.S. Cl.** **709/230**

[58] **Field of Search** 364/131, 138; 345/333–335; 709/203, 217, 219, 223, 224, 230

[56] **References Cited**

U.S. PATENT DOCUMENTS

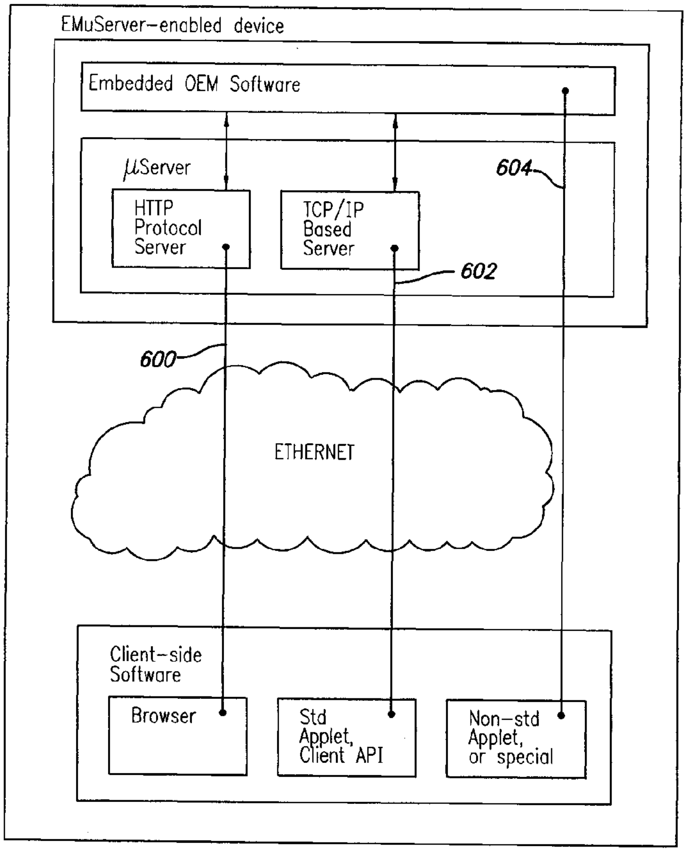
4,860,216	8/1989	Linsenmayer .	
4,901,218	2/1990	Cornwell .	
5,428,555	6/1995	Starkey et al. .	
5,472,347	12/1995	Nordenstrom et al. .	
5,512,890	4/1996	Everson, Jr. et al. .	
5,528,219	6/1996	Frohlich et al. .	
5,598,521	1/1997	Kilgore et al. .	
5,664,101	9/1997	Picache .	
5,794,032	8/1998	Leyda	713/2
5,805,442	9/1998	Crater et al.	364/138

Primary Examiner—Zarni Maung
Attorney, Agent, or Firm—Banner & Witcoff, Ltd.

[57] **ABSTRACT**

A micro-server adapted to be embedded into a piece of industrial machinery, an automobile, a consumer product, and the like, for publishing information, possibly in the form of web pages, about the device into which the micro-server is embedded or with which it is associated and/or for controlling a micro-server equipped device from a possibly remote client. The information may be published such that it is accessible using a standard web-browser. Other suitable protocols could also be used. The micro-server is capable of interfacing with a device to access information from the device, such as control or maintenance information. The micro-server can then organize and format that information compatible with a communication protocol in preparation for publishing the information. The micro-server conveniently abstracts from the first device the details of the communication protocol used to publish the information.

35 Claims, 17 Drawing Sheets



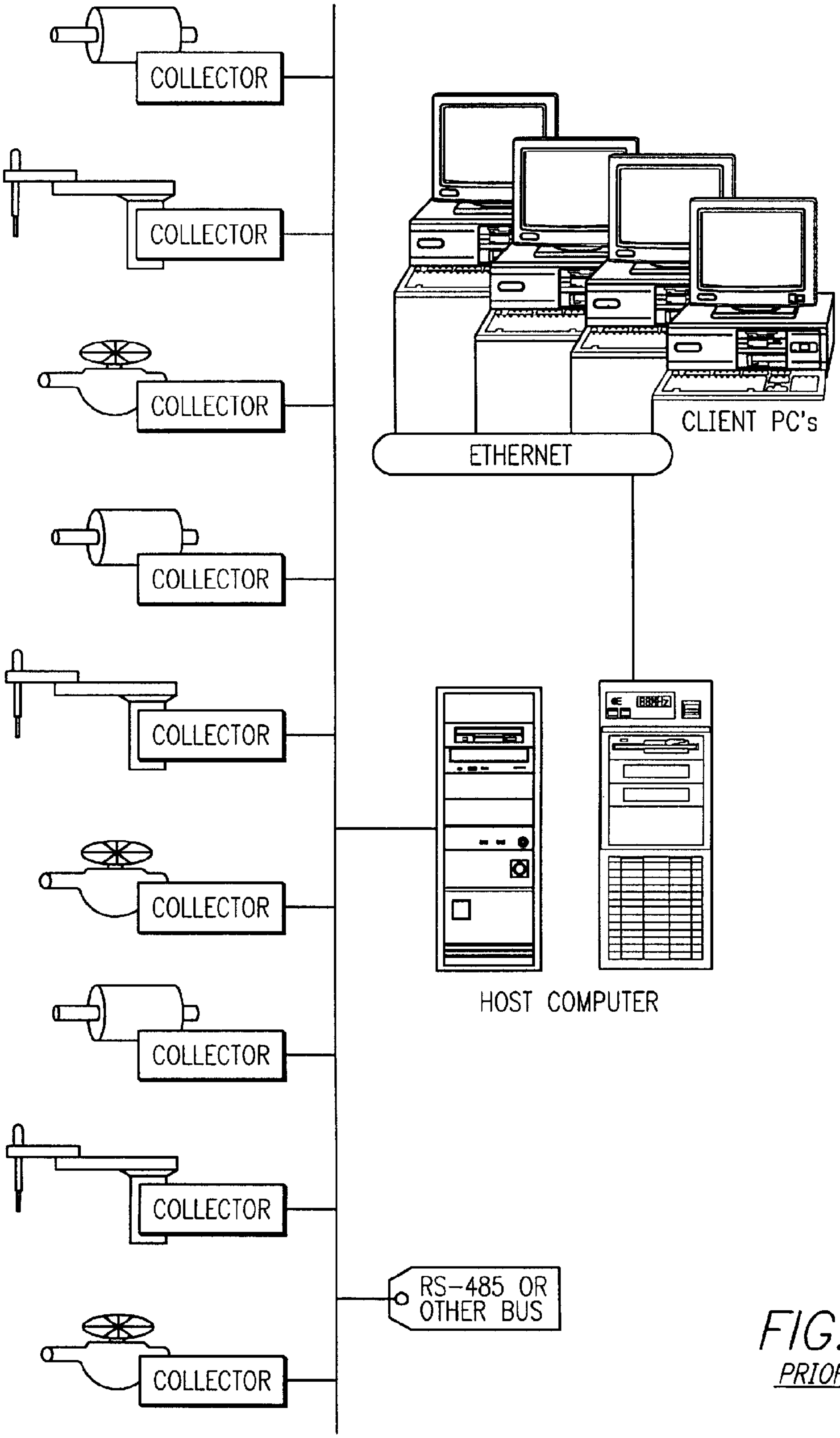


FIG. 1
PRIOR ART

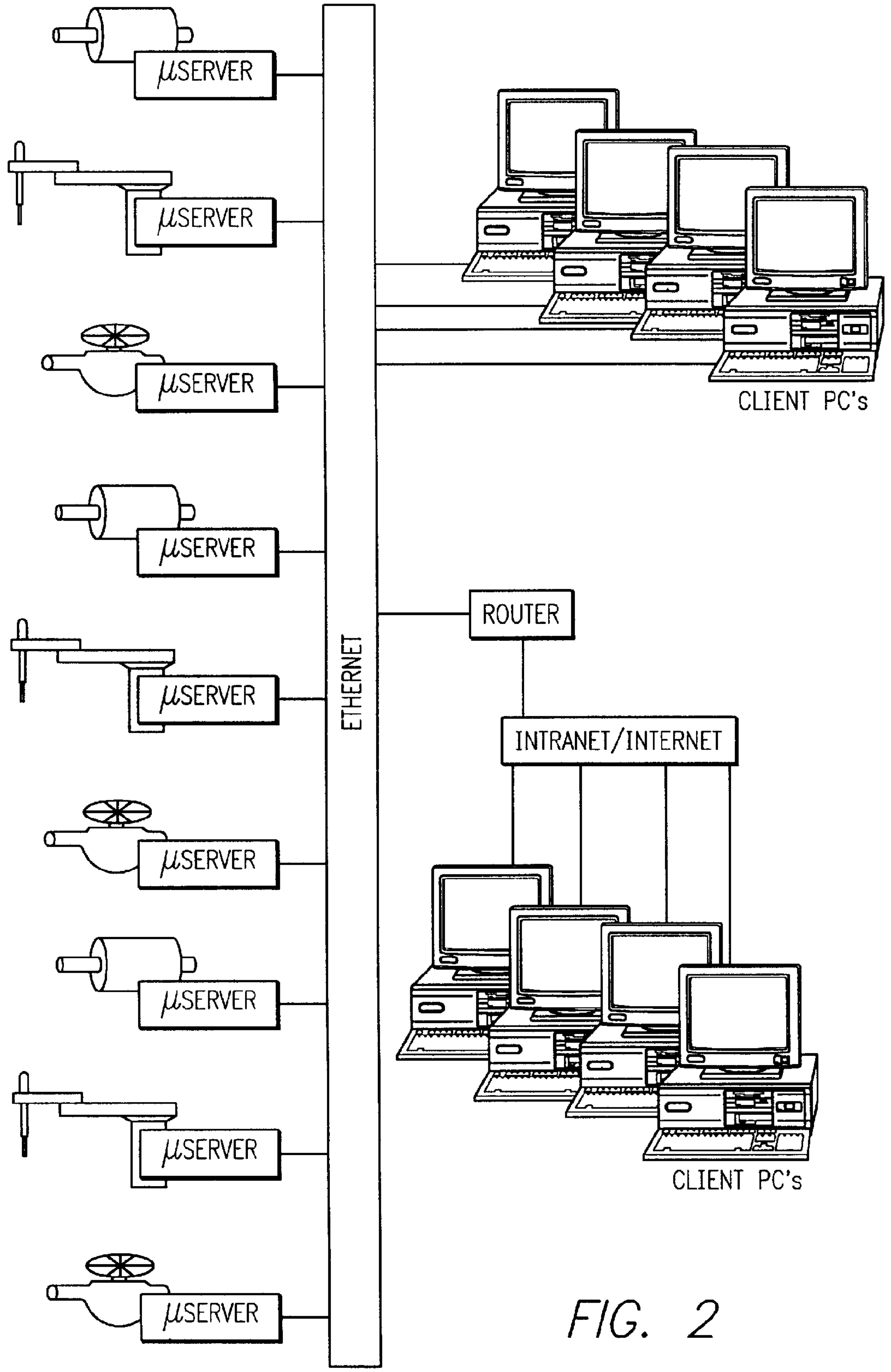

μSERVER
μSERVER
μSERVER
μSERVER
μSERVER
μSERVER
μSERVER
μSERVER
μSERVER
ETHERNET
CLIENT PC's
ROUTER
INTRANET/INTERNET
CLIENT PC's

FIG. 2

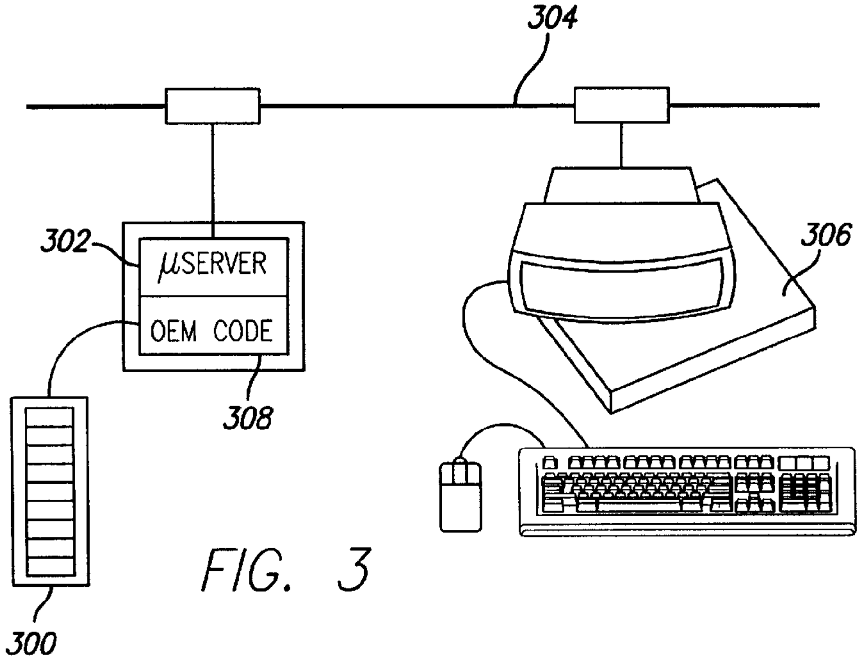

304
302
μSERVER
OEM CODE
308
306
300

FIG. 3

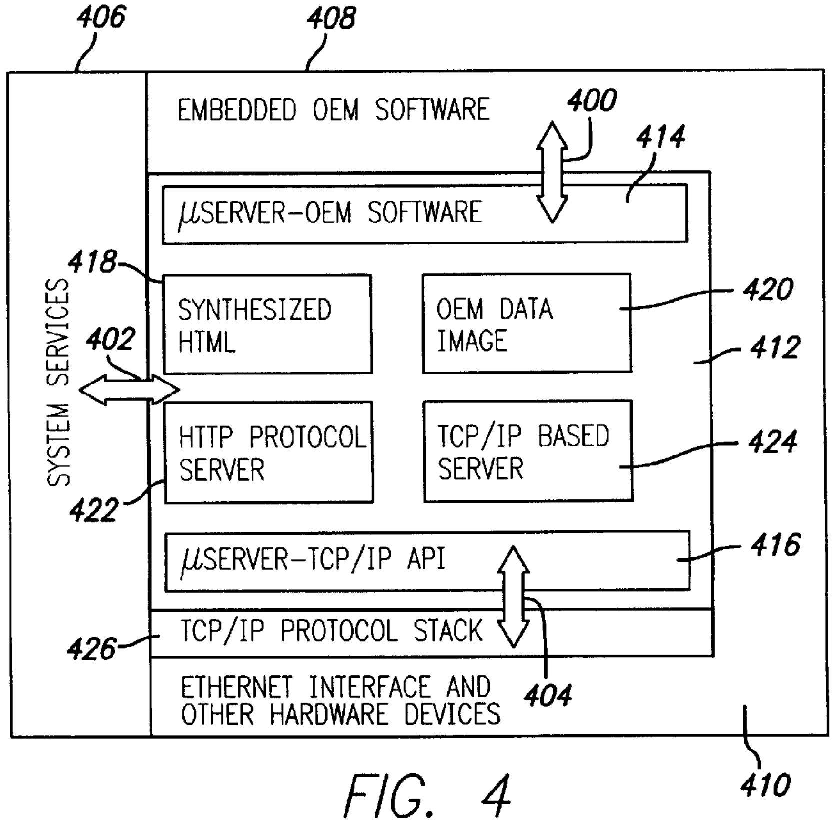

406
408
EMBEDDED OEM SOFTWARE
400
414
μSERVER-OEM SOFTWARE
418
SYNTHESIZED HTML
OEM DATA IMAGE
420
412
SYSTEM SERVICES
402
HTTP PROTOCOL SERVER
TCP/IP BASED SERVER
424
422
μSERVER-TCP/IP API
416
426
TCP/IP PROTOCOL STACK
ETHERNET INTERFACE AND OTHER HARDWARE DEVICES
404
410

FIG. 4

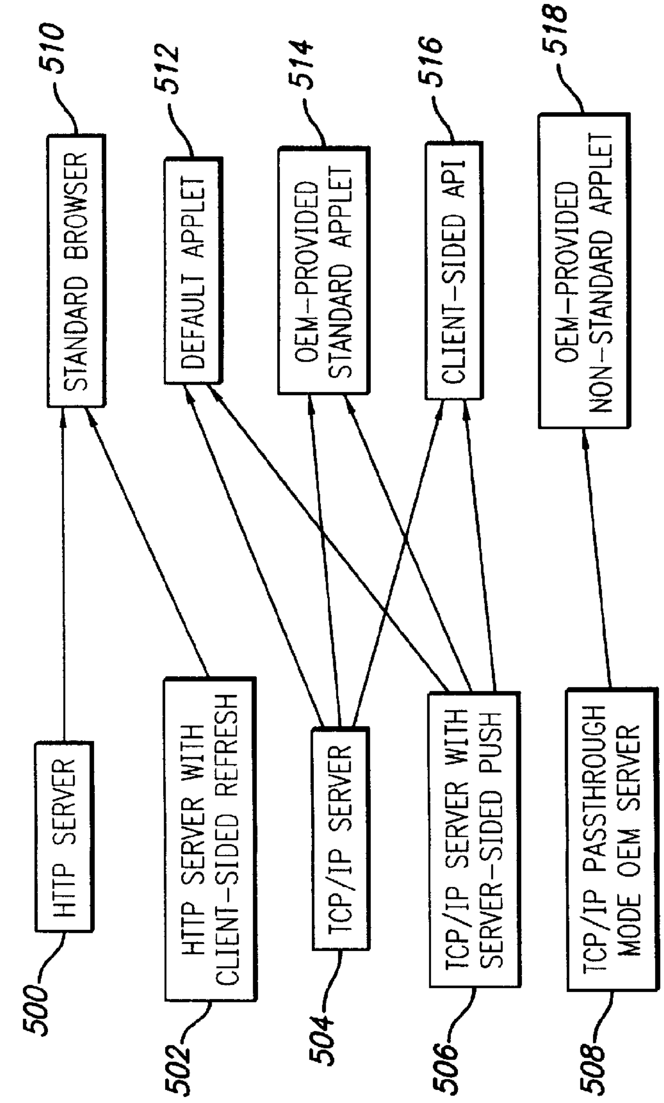

500
HTTP SERVER
502
HTTP SERVER WITH CLIENT-SIDED REFRESH
504
TCP/IP SERVER
506
TCP/IP SERVER WITH SERVER-SIDED PUSH
508
TCP/IP PASSTHROUGH MODE OEM SERVER
510
STANDARD BROWSER
512
DEFAULT APPLET
514
OEM-PROVIDED STANDARD APPLET
516
CLIENT-SIDED API
518
OEM-PROVIDED NON-STANDARD APPLET

FIG. 5

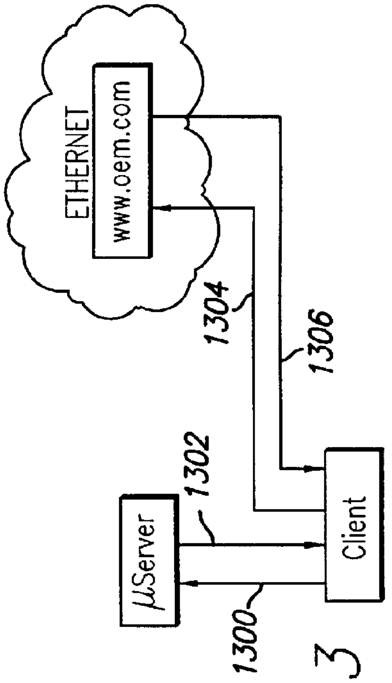

ETHERNET
www.oem.com
1304
1306
1302
μServer
Client
1300

FIG. 13

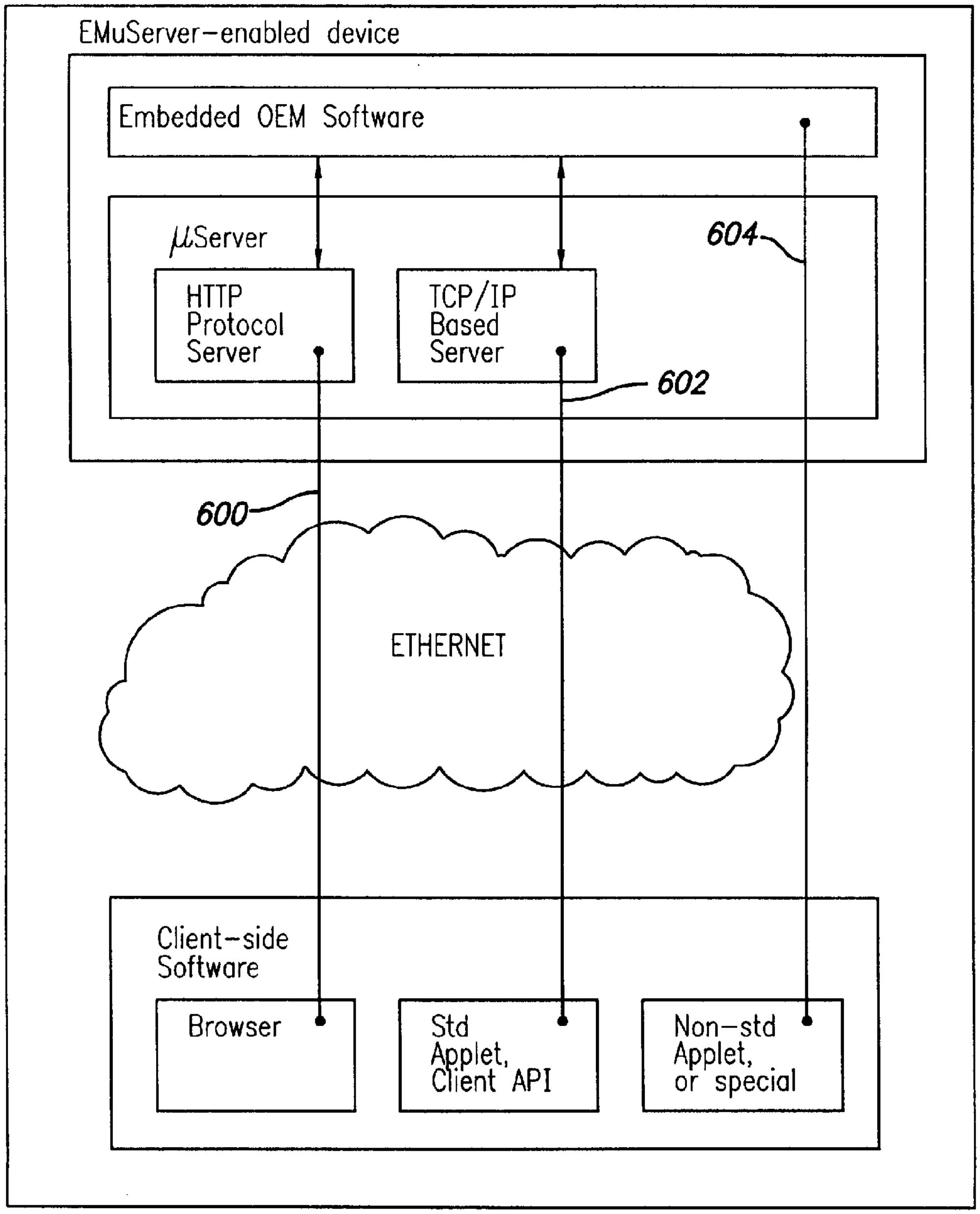
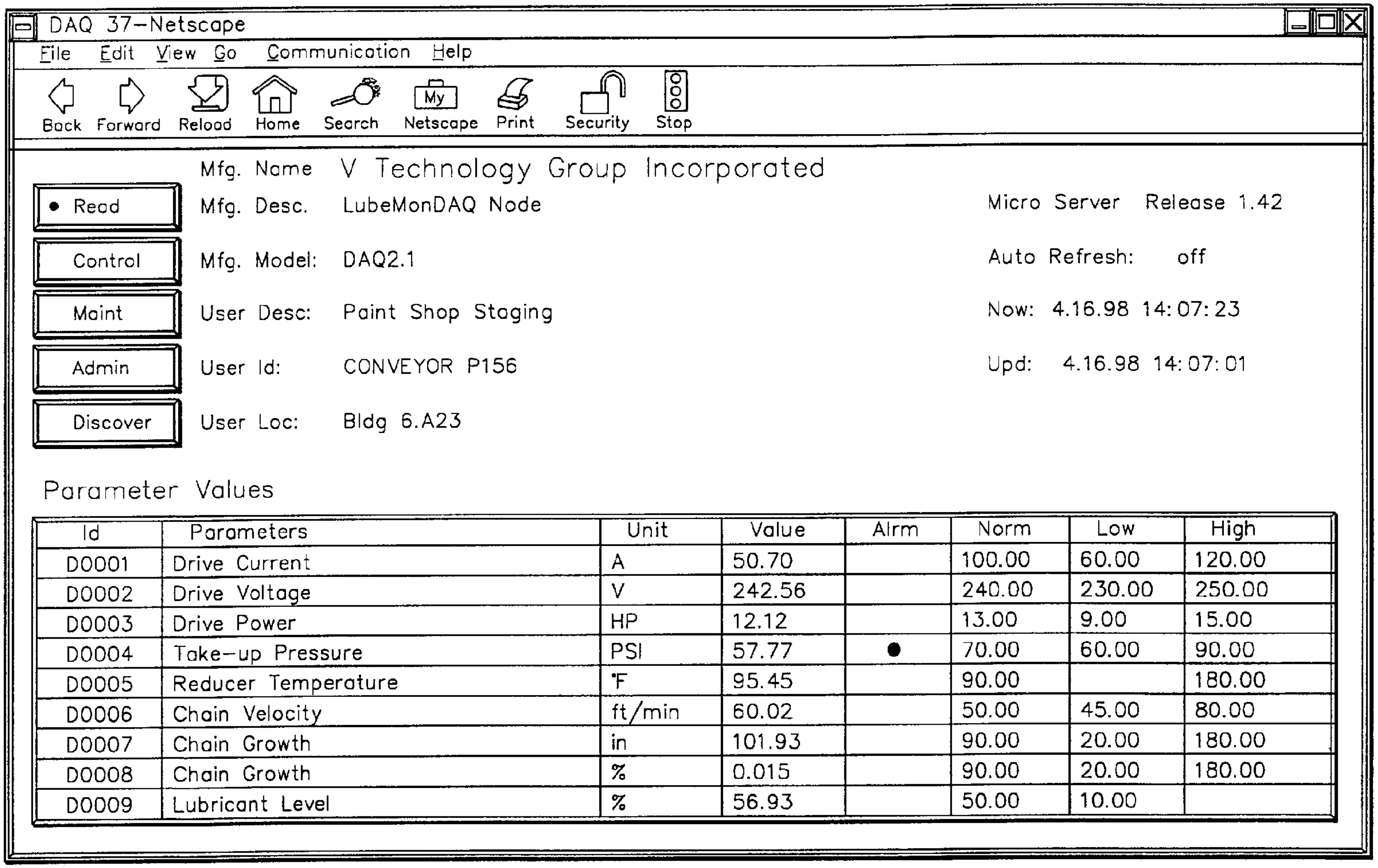

EMuServer–enabled device
Embedded OEM Software
µServer
604
HTTP Protocol Server
TCP/IP Based Server
602
600
ETHERNET
Client–side Software
Browser
Std Applet, Client API
Non–std Applet, or special

FIG. 6

FIG. 7



Mfg. Name V Technology Group Incorporated

Mfg. Desc. LubeMonDAQ Node

Mfg. Model: DAQ2.1

User Desc: Paint Shop Staging

User Id: CONVEYOR P156

User Loc: Bldg 6.A23

Micro Server Release 1.42

Auto Refresh: off

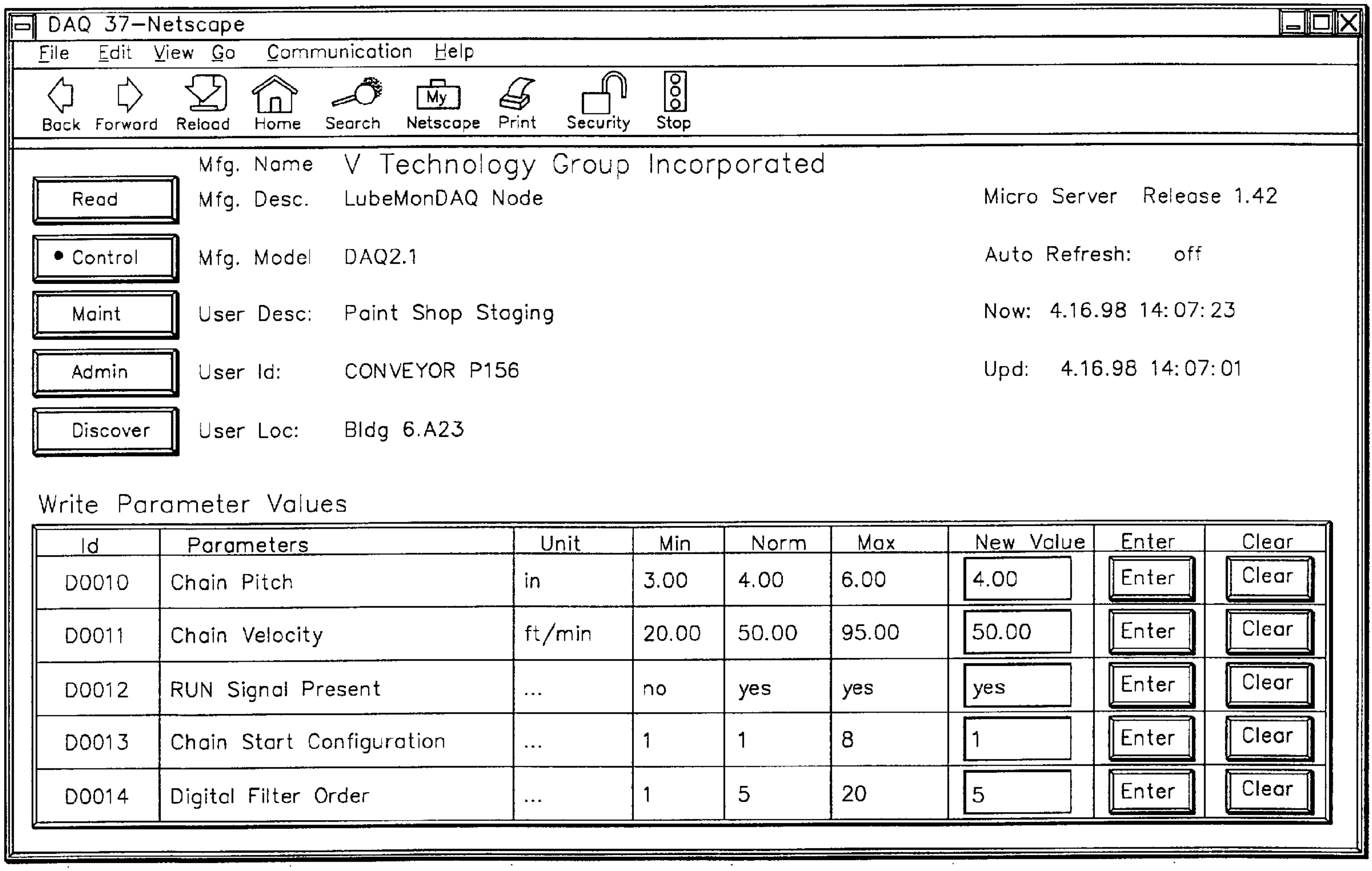
Now: 4.16.98 14:07:23

Upd: 4.16.98 14:07:01

Parameter Values

Id	Parameters	Unit	Value	Alrm	Norm	Low	High
D0001	Drive Current	A	50.70		100.00	60.00	120.00
D0002	Drive Voltage	V	242.56		240.00	230.00	250.00
D0003	Drive Power	HP	12.12		13.00	9.00	15.00
D0004	Take-up Pressure	PSI	57.77	●	70.00	60.00	90.00
D0005	Reducer Temperature	°F	95.45		90.00		180.00
D0006	Chain Velocity	ft/min	60.02		50.00	45.00	80.00
D0007	Chain Growth	in	101.93		90.00	20.00	180.00
D0008	Chain Growth	%	0.015		90.00	20.00	180.00
D0009	Lubricant Level	%	56.93		50.00	10.00	

FIG. 8



Mfg. Name V Technology Group Incorporated

Mfg. Desc. LubeMonDAQ Node

Mfg. Model DAQ2.1

User Desc: Paint Shop Staging

User Id: CONVEYOR P156

User Loc: Bldg 6.A23

Micro Server Release 1.42

Auto Refresh: off

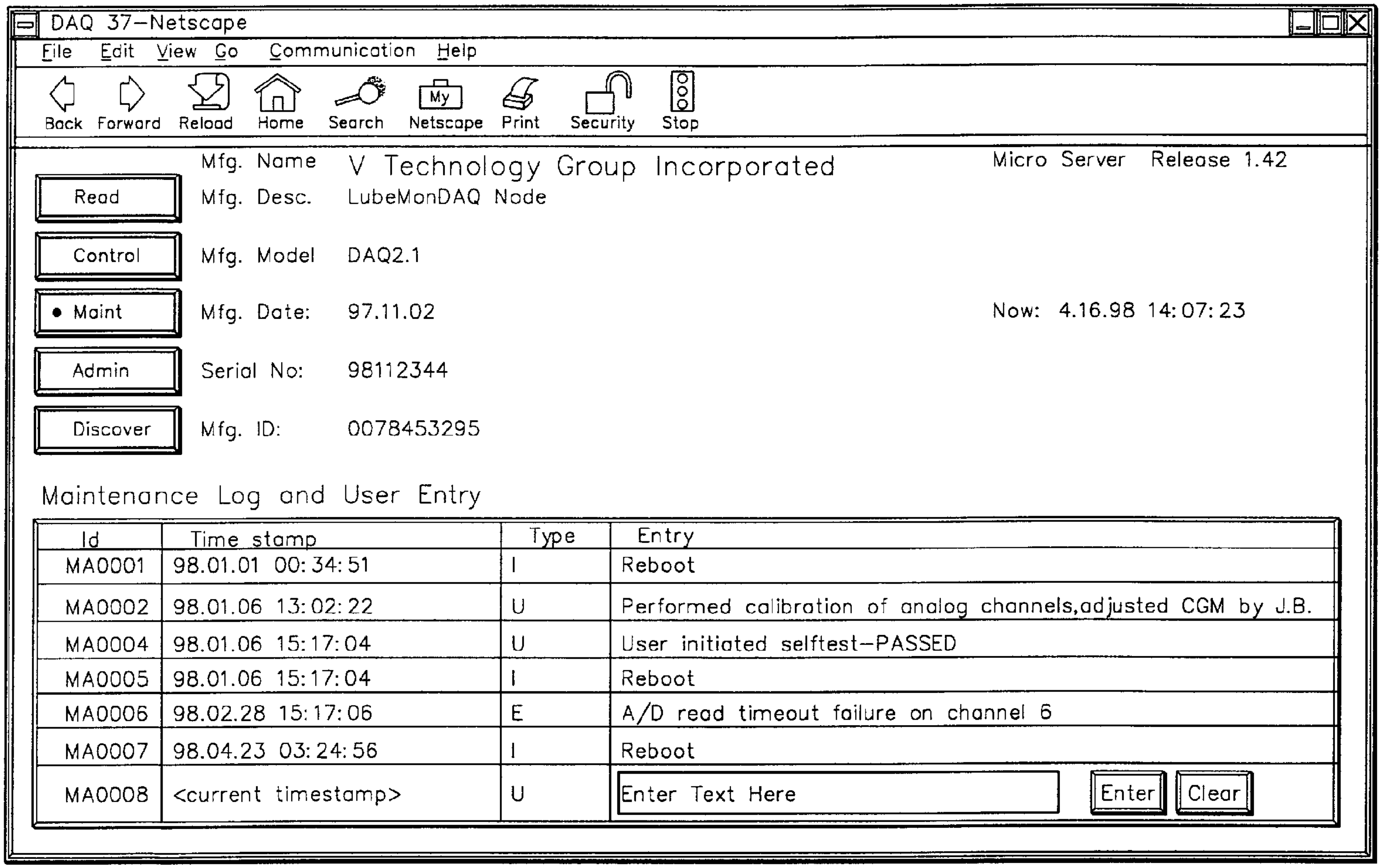
Now: 4.16.98 14:07:23

Upd: 4.16.98 14:07:01

Write Parameter Values

Id	Parameters	Unit	Min	Norm	Max	New Value	Enter	Clear
D0010	Chain Pitch	in	3.00	4.00	6.00	4.00	Enter	Clear
D0011	Chain Velocity	ft/min	20.00	50.00	95.00	50.00	Enter	Clear
D0012	RUN Signal Present	...	no	yes	yes	yes	Enter	Clear
D0013	Chain Start Configuration	...	1	1	8	1	Enter	Clear
D0014	Digital Filter Order	...	1	5	20	5	Enter	Clear

FIG. 9



Mfg. Name V Technology Group Incorporated

Micro Server Release 1.42

Mfg. Desc. LubeMonDAQ Node

Mfg. Model DAQ2.1

Mfg. Date: 97.11.02

Now: 4.16.98 14:07:23

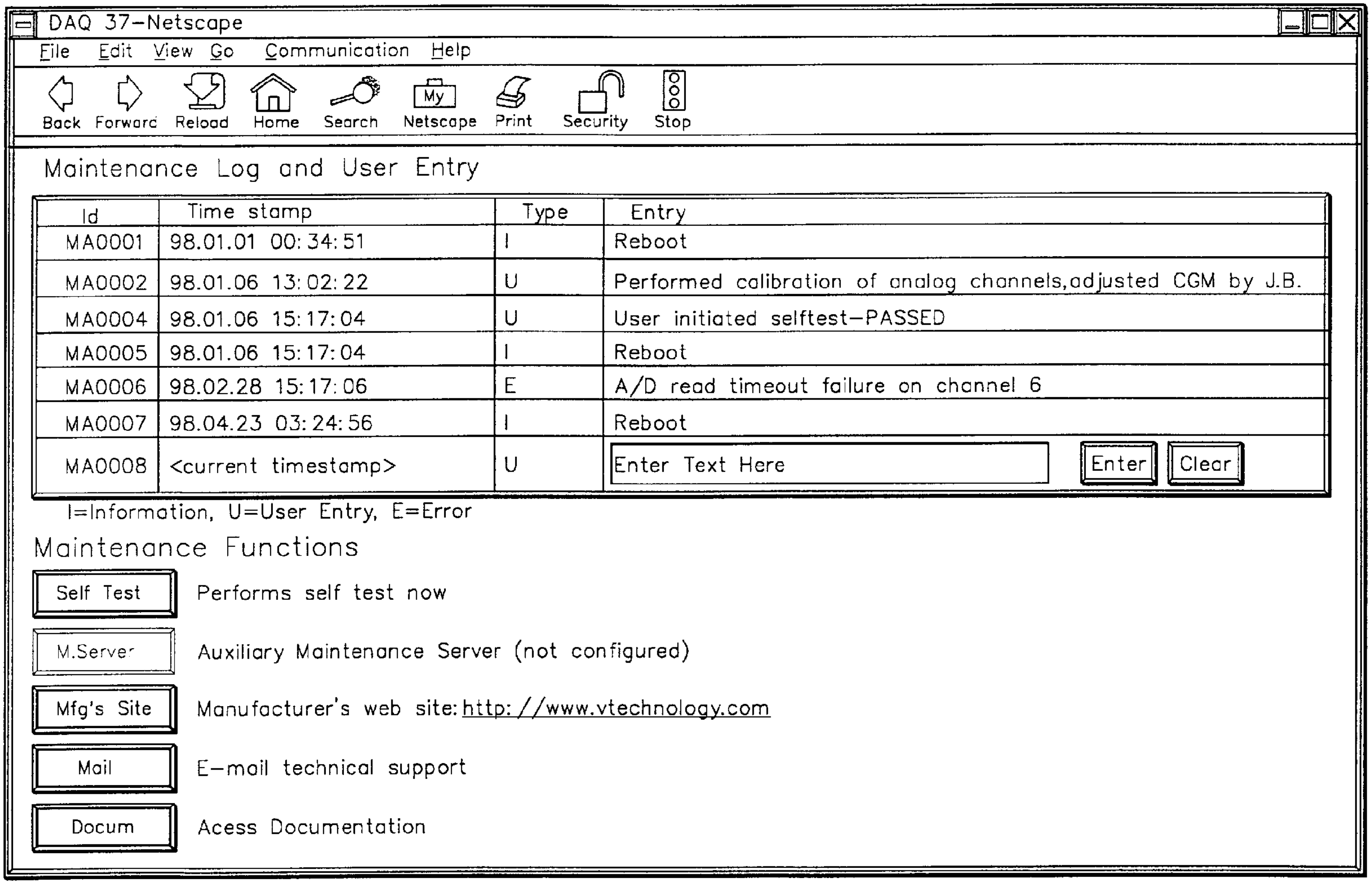
Serial No: 98112344

Mfg. ID: 0078453295

Maintenance Log and User Entry

Id	Time stamp	Type	Entry
MA0001	98.01.01 00:34:51	I	Reboot
MA0002	98.01.06 13:02:22	U	Performed calibration of analog channels,adjusted CGM by J.B.
MA0004	98.01.06 15:17:04	U	User initiated selftest–PASSED
MA0005	98.01.06 15:17:04	I	Reboot
MA0006	98.02.28 15:17:06	E	A/D read timeout failure on channel 6
MA0007	98.04.23 03:24:56	I	Reboot
MA0008	<current timestamp>	U	Enter Text Here

FIG. 10


DAQ 37–Netscape
File Edit View Go Communication Help
Back Forward Reload Home Search My Netscape Print Security Stop

Maintenance Log and User Entry

Id	Time stamp	Type	Entry
MA0001	98.01.01 00:34:51	I	Reboot
MA0002	98.01.06 13:02:22	U	Performed calibration of analog channels,adjusted CGM by J.B.
MA0004	98.01.06 15:17:04	U	User initiated selftest–PASSED
MA0005	98.01.06 15:17:04	I	Reboot
MA0006	98.02.28 15:17:06	E	A/D read timeout failure on channel 6
MA0007	98.04.23 03:24:56	I	Reboot
MA0008	<current timestamp>	U	Enter Text Here Enter Clear

I=Information, U=User Entry, E=Error

Maintenance Functions

Self Test Performs self test now

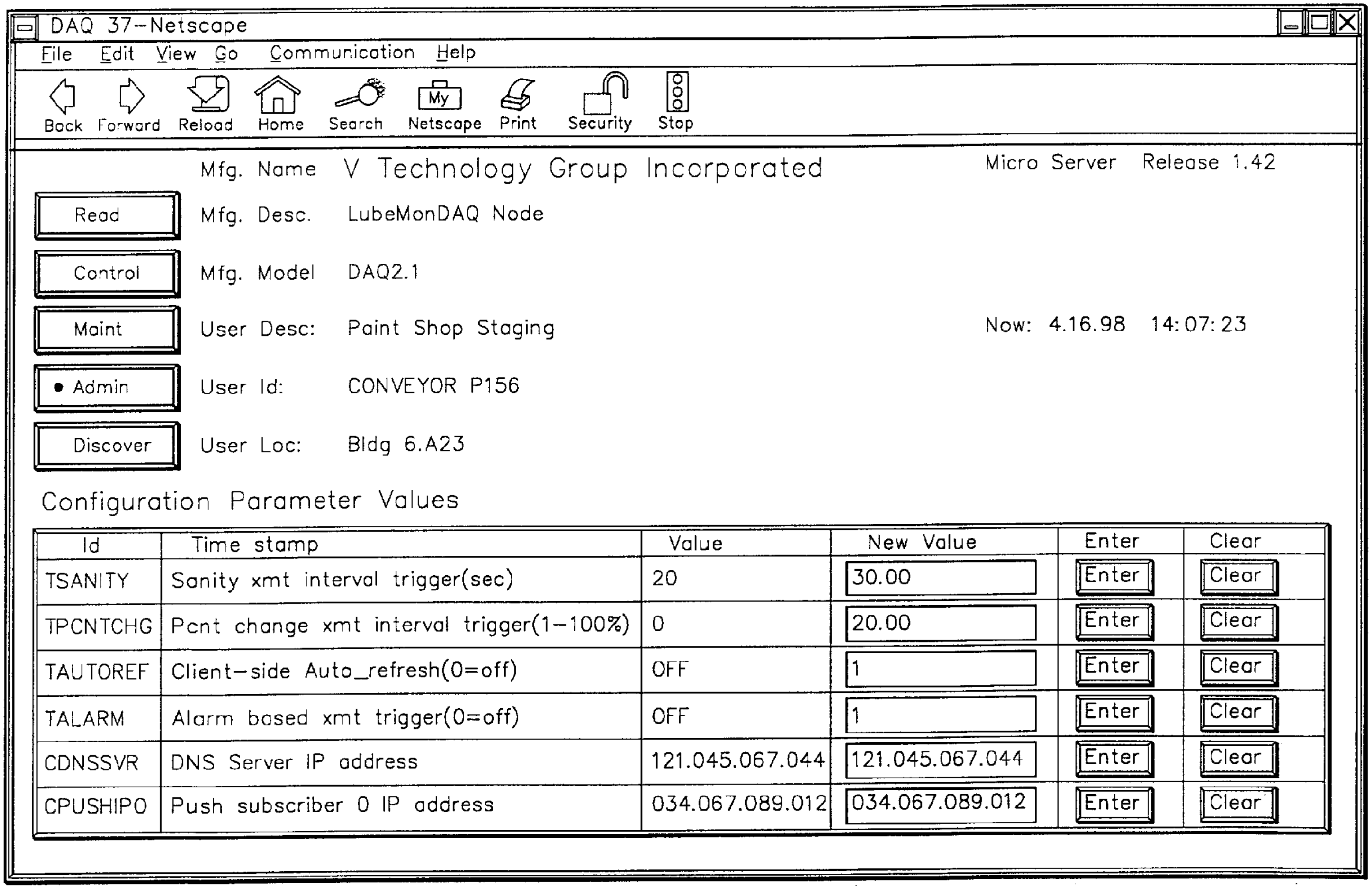
M.Server Auxiliary Maintenance Server (not configured)

Mfg's Site Manufacturer's web site: http://www.vtechnology.com

Mail E–mail technical support

Docum Acess Documentation

FIG. 11



Mfg. Name V Technology Group Incorporated

Micro Server Release 1.42

Mfg. Desc. LubeMonDAQ Node

Mfg. Model DAQ2.1

User Desc: Paint Shop Staging

Now: 4.16.98 14:07:23

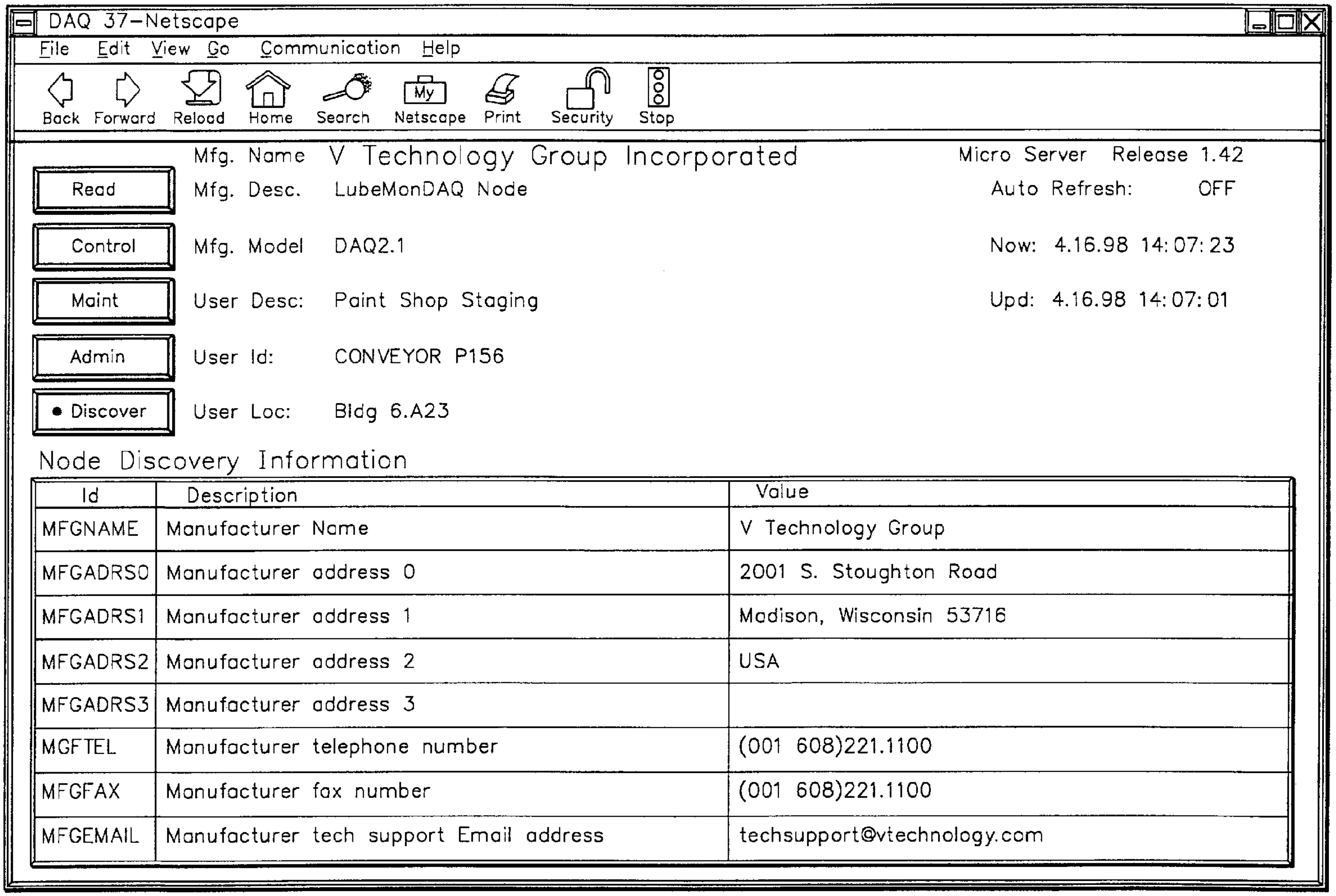
User Id: CONVEYOR P156

User Loc: Bldg 6.A23

Configuration Parameter Values

Id	Time stamp	Value	New Value	Enter	Clear
TSANITY	Sanity xmt interval trigger(sec)	20	30.00	Enter	Clear
TPCNTCHG	Pcnt change xmt interval trigger(1–100%)	0	20.00	Enter	Clear
TAUTOREF	Client–side Auto_refresh(0=off)	OFF	1	Enter	Clear
TALARM	Alarm based xmt trigger(0=off)	OFF	1	Enter	Clear
CDNSSVR	DNS Server IP address	121.045.067.044	121.045.067.044	Enter	Clear
CPUSHIPO	Push subscriber 0 IP address	034.067.089.012	034.067.089.012	Enter	Clear

FIG. 12



Mfg. Name V Technology Group Incorporated

Mfg. Desc. LubeMonDAQ Node

Mfg. Model DAQ2.1

User Desc: Paint Shop Staging

User Id: CONVEYOR P156

User Loc: Bldg 6.A23

Micro Server Release 1.42

Auto Refresh: OFF

Now: 4.16.98 14:07:23

Upd: 4.16.98 14:07:01

Node Discovery Information

Id	Description	Value
MFGNAME	Manufacturer Name	V Technology Group
MFGADRS0	Manufacturer address 0	2001 S. Stoughton Road
MFGADRS1	Manufacturer address 1	Madison, Wisconsin 53716
MFGADRS2	Manufacturer address 2	USA
MFGADRS3	Manufacturer address 3	
MGFTEL	Manufacturer telephone number	(001 608)221.1100
MFGFAX	Manufacturer fax number	(001 608)221.1100
MFGEMAIL	Manufacturer tech support Email address	techsupport@vtechnology.com

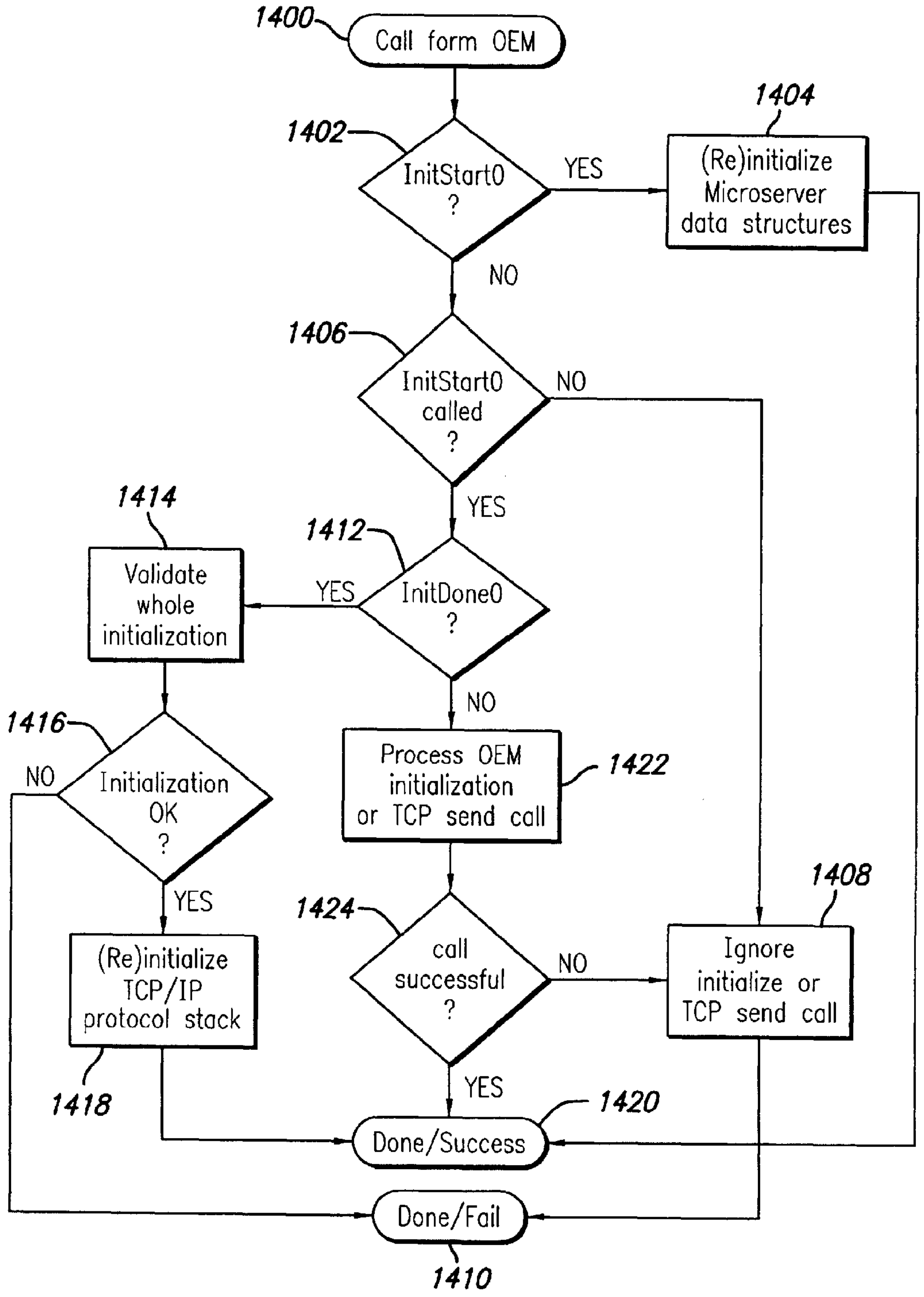

1400
Call form OEM
1402
InitStart0 ?
YES
1404
(Re)initialize Microserver data structures
NO
1406
InitStart0 called ?
NO
YES
1414
1412
Validate whole initialization
YES
InitDone0 ?
NO
1416
NO
Initialization OK ?
Process OEM initialization or TCP send call
1422
YES
1408
1424
(Re)initialize TCP/IP protocol stack
call successful ?
NO
Ignore initialize or TCP send call
1418
YES
1420
Done/Success
Done/Fail
1410

FIG. 14

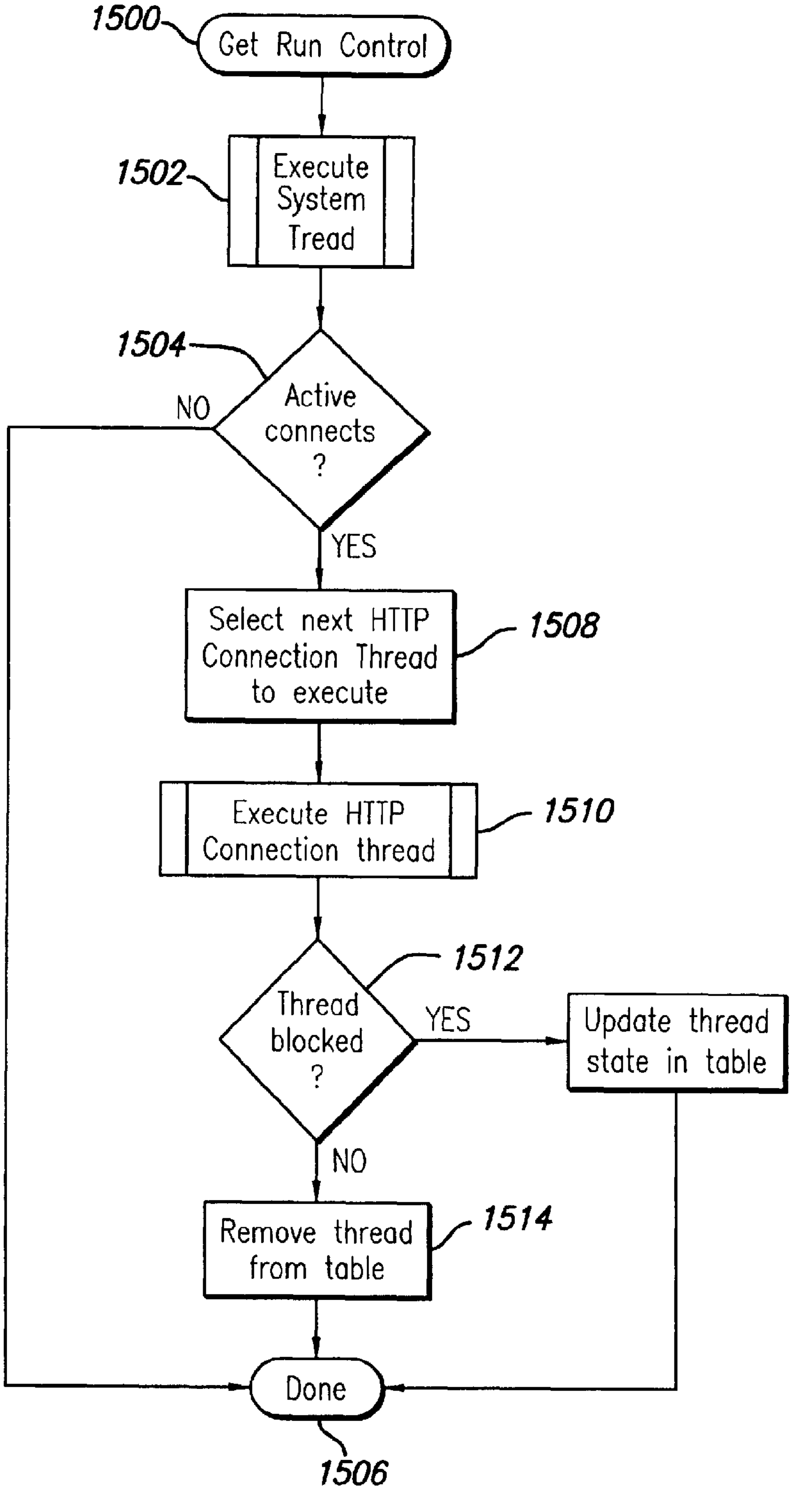

1500
Get Run Control
1502
Execute System Tread
1504
Active connects ?
NO
YES
Select next HTTP Connection Thread to execute
1508
Execute HTTP Connection thread
1510
1512
Thread blocked ?
YES
Update thread state in table
NO
Remove thread from table
1514
Done
1506

FIG. 15

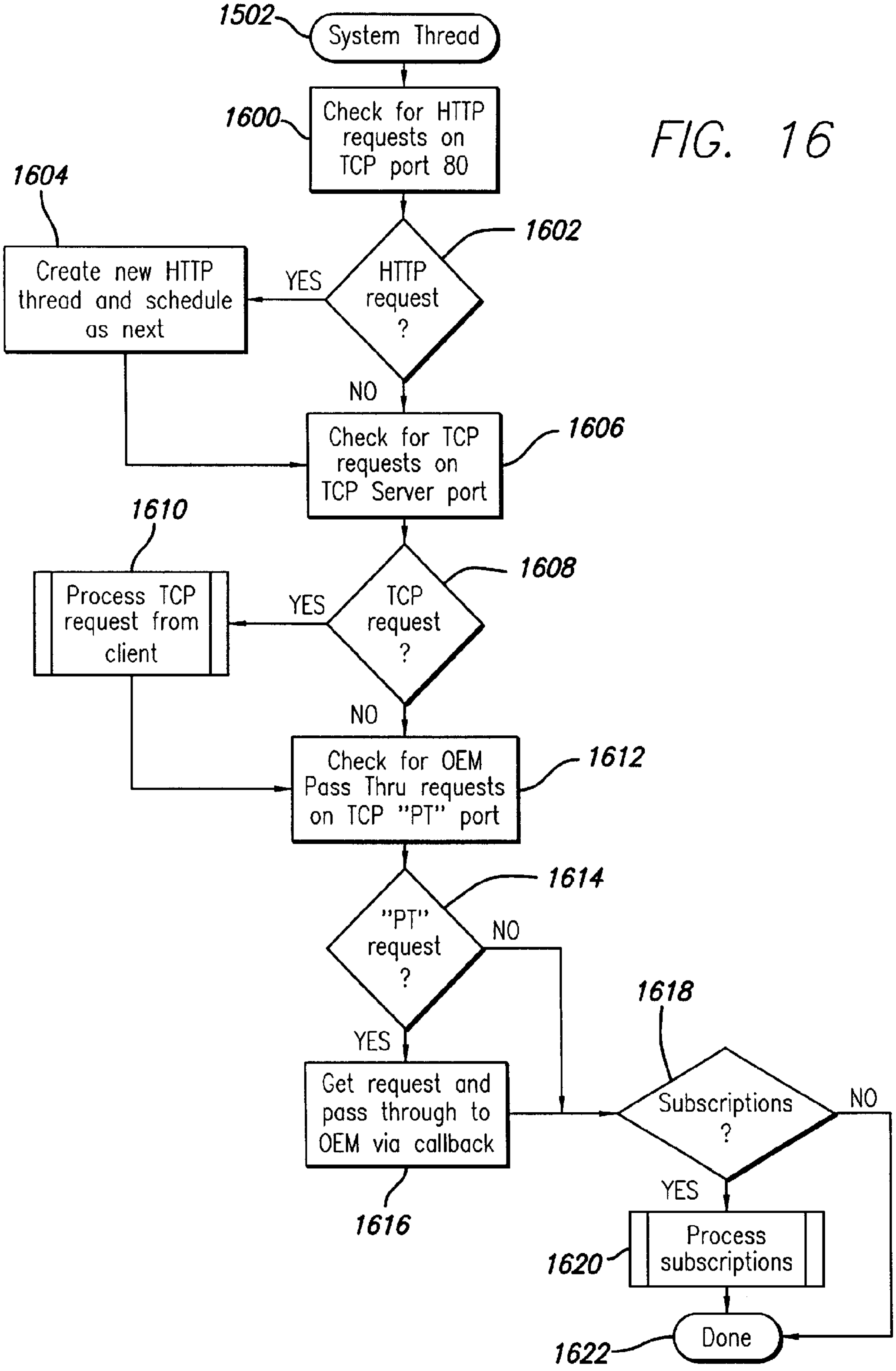

1502
System Thread
1600
Check for HTTP requests on TCP port 80
1602
HTTP request ?
YES
NO
1604
Create new HTTP thread and schedule as next
1606
Check for TCP requests on TCP Server port
1608
TCP request ?
YES
NO
1610
Process TCP request from client
1612
Check for OEM Pass Thru requests on TCP "PT" port
1614
"PT" request ?
NO
YES
1616
Get request and pass through to OEM via callback
1618
Subscriptions ?
NO
YES
1620
Process subscriptions
1622
Done

FIG. 16

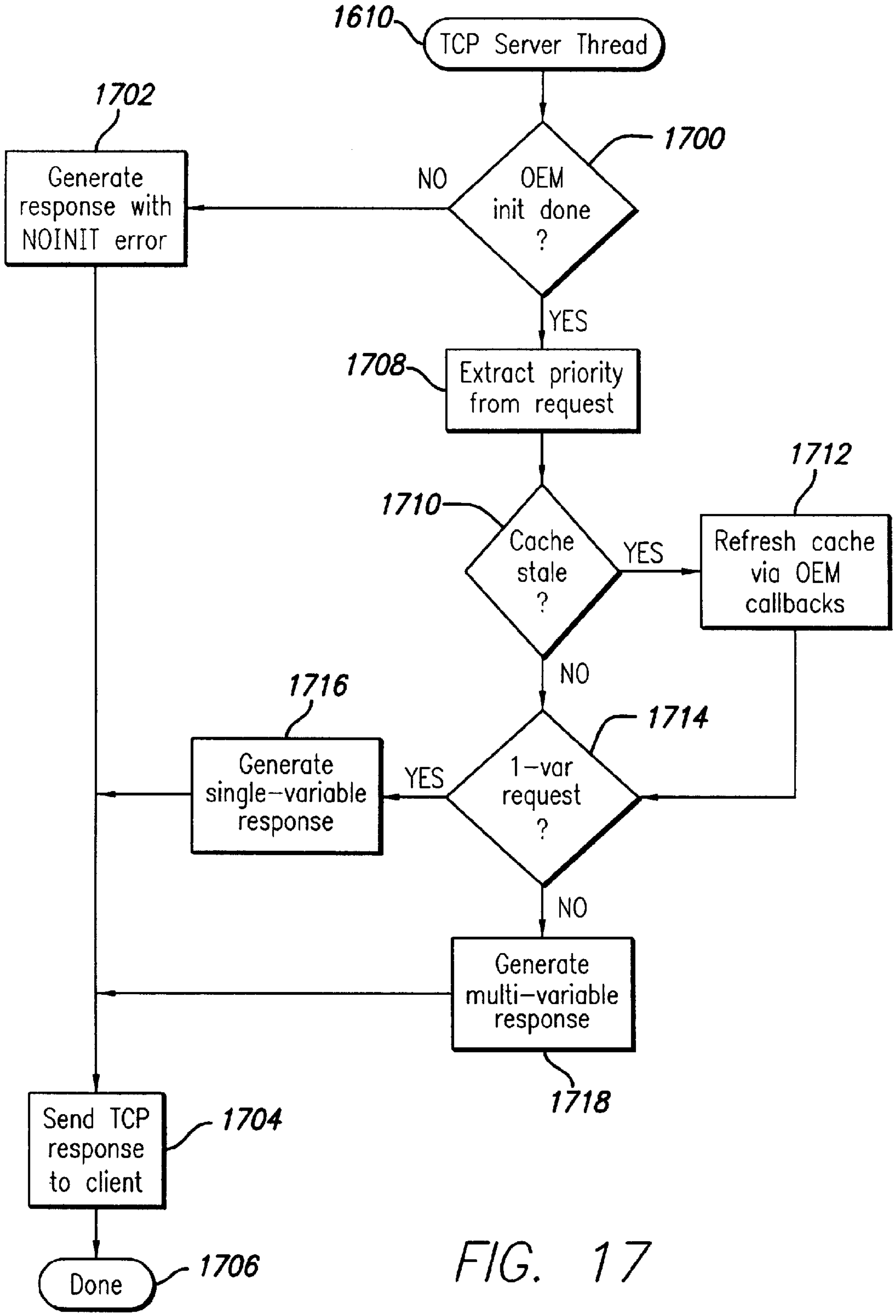

1610
TCP Server Thread
1700
OEM init done ?
NO
1702
Generate response with NOINIT error
YES
1708
Extract priority from request
1710
Cache stale ?
YES
1712
Refresh cache via OEM callbacks
NO
1714
1-var request ?
YES
1716
Generate single-variable response
NO
Generate multi-variable response
1718
Send TCP response to client
1704
Done
1706

FIG. 17

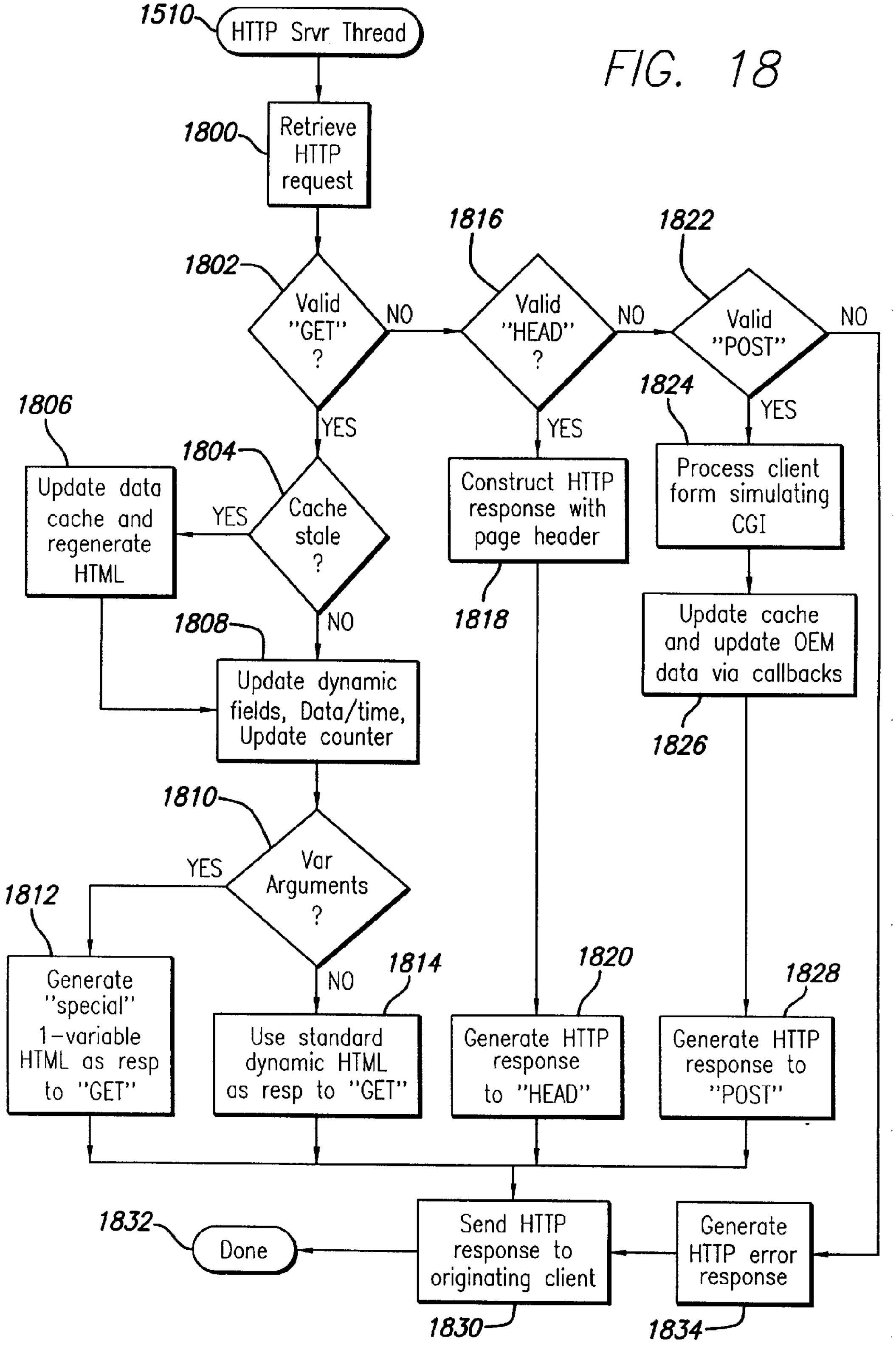
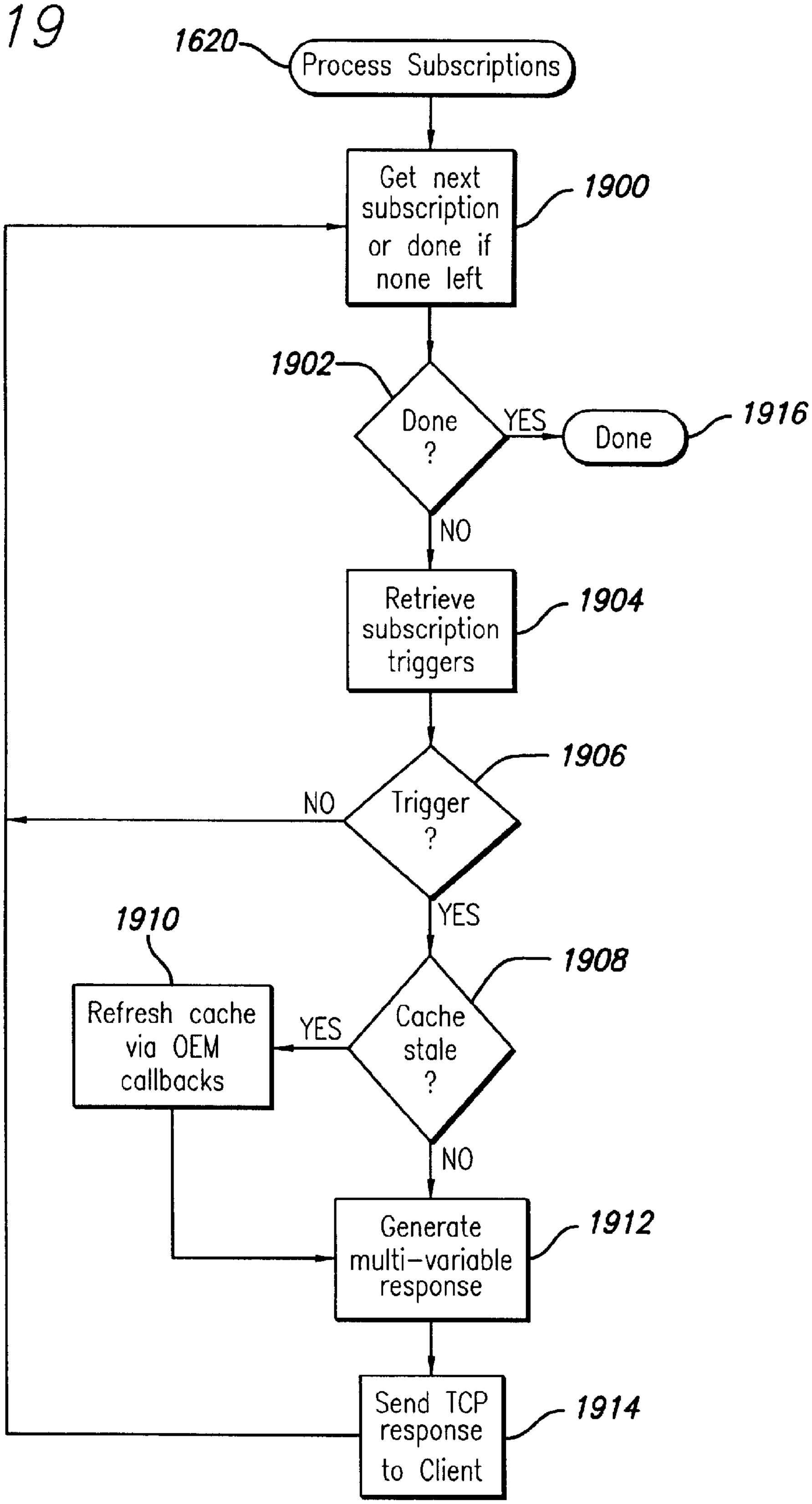

1510
HTTP Srvr Thread
1800
Retrieve HTTP request
1802
Valid "GET" ?
NO
YES
1816
Valid "HEAD" ?
NO
YES
1822
Valid "POST" ?
NO
YES
1806
Update data cache and regenerate HTML
1804
YES
Cache stale ?
NO
Construct HTTP response with page header
1818
1824
Process client form simulating CGI
Update cache and update OEM data via callbacks
1826
1808
Update dynamic fields, Data/time, Update counter
1810
YES
Var Arguments ?
NO
1812
Generate "special" 1-variable HTML as resp to "GET"
1814
Use standard dynamic HTML as resp to "GET"
1820
Generate HTTP response to "HEAD"
1828
Generate HTTP response to "POST"
1832
Done
Send HTTP response to originating client
1830
Generate HTTP error response
1834

FIG. 18

FIG. 19


1620
Process Subscriptions
Get next subscription or done if none left
1900
1902
Done ?
YES
Done
1916
NO
Retrieve subscription triggers
1904
1906
Trigger ?
NO
YES
1910
Refresh cache via OEM callbacks
YES
Cache stale ?
1908
NO
Generate multi-variable response
1912
Send TCP response to Client
1914

SYSTEM AND METHOD FOR ACCESSING INFORMATION FROM A REMOTE DEVICE AND PROVIDING THE INFORMATION TO A CLIENT WORKSTATION

BACKGROUND OF THE INVENTION

1. Field of the Invention

This invention relates to providing information from a first device to a second device. More particularly, this invention relates to a micro-server system comprising a micro-server capable of being embedded into and/or associated with industrial equipment and/or consumer devices/products and capable of publishing information about such equipment and/or devices/products to thin clients running standard web browser software.

2. Statement of Related Art

Industrial process information is typically collected and used primarily through the offerings of a handful of key industrial information collection companies, or through internal home-brew solutions. Both approaches are costly to implement because the collecting architecture is very specific to the individual devices and to the whole process. Almost all the transport protocols are proprietary, and much of the media used to interconnect these devices, like RS-485, DeviceNet, and the like, are either proprietary or are limited to connecting only these kinds of devices. For a very remotely located device, options for connection can be severely limited.

In addition, most prior art solutions are hard-wired to the process, and a central host collects and manages device data, as is shown in FIG. **1**. A central host is not a natural place for this information. The data is more timely, accurate, and meaningful at the device to which the data pertains.

Should the process change, re-wiring or re-instrumenting of prior art systems is typically needed. Re-programming the host is an enormous task fraught with the potential for bringing the whole process to a halt. The proprietary software that communicates with the host is usually licensed for, and installed on, each client computer, representing a big investment even from the update management perspective alone.

A significant drawback of such prior art systems is that if the host fails or is unreachable for any reason, all tactical and strategic data becomes unavailable.

Therefore, it is an object of this invention to simultaneously remove the host computer, as shown in FIG. **2** in which micro-server is abbreviated μServer, remove the need for customer programming, unify the network fabric throughout factories and offices (Ethernet), provide secure access to any effector or device in the process from any workstation in the enterprise, and reduce the total cost of ownership.

It is an additional object of this invention to prevent information about a device from being maintained on a centralized host computer and to allow such information to reside in the actual device itself. With the information residing in the device itself, if the device is moved, its data moves with it. If the device is replaced, the new device can automatically publish its new data according to the principles of this invention.

It is a further object of this invention to enable devices to come on line and be browsable by a browser when the devices are shipped from the OEM. According to the principles of this invention, such devices could be capable of providing operational data, limits, suggested maintenance cycles, specifications, links to the manufacturer's web site for detailed drawings, and literally whatever other information that the OEM desires.

Typically, original equipment manufacturer ("OEM") software professionals do not program with the Windows Application Programming Interface ("API"), and therefore almost never write code for a network. Typically, however, they are very familiar with the embedded software necessary to monitor and control industrial devices.

It is therefore a further object of this invention to abstract and encapsulate the highly complex TCP/IP network layer and Internet Web services to provide a simple, yet comprehensive, API in the "embedded" problem space, with which most OEM software professionals are familiar. It is a further object of this invention to publish information about the industrial equipment, also referred to as the device data, to an enterprise or the world as a web page on the corporate Intranet or the larger Internet.

SUMMARY OF THE INVENTION

A system for providing information about a first device to a second device. A micro-server interfaces with the first device to access the information from the first device. The information is then organized and formatted compatible with a communication protocol in preparation for making the information available to said second device. The information is made available to the second device while abstracting the communication protocol from the first device.

The system could include: an OEM application programming interface ("API") for interfacing between the first device's software layer and the micro-server; a TCP/IP stack for interfacing with a hardware interface; a TCP/IP API for providing access to the TCP/IP protocol stack; a hardware Ethernet interface; a system services API for providing the micro-server access to system services from the first device; an HTTP protocol server for satisfying interactive HTTP requests; a browser for interacting with the first device; a TCP/IP-based server for satisfying TCP/IP-based requests; a hyperlink to a website associated with the first device; a web-site associated with the first device; a default applet server for providing default applets to the second device to interface with the first device; a time server for providing current time information; an auto-discovery and view server for automatically detecting the first device being coupled to an interface; and a browser for interacting with the first device.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** illustrates a prior art architecture including a centralized host computer for collecting industrial process information.

FIG. **2** illustrates a possible architecture, consistent with the principles of this invention, for publishing industrial process information.

FIG. **3** illustrates a simplified example system in accordance with the principles of this invention.

FIG. **4** is a simplified block diagram showing several subsystems of a micro-server according to this invention.

FIG. **5** illustrates several possible server/client types of combinations according to this invention.

FIG. **6** illustrates three primary communication paths between a micro-server-enabled device and client-side software.

FIG. **7** is a sample Read (Default) page.

FIG. **8** is a sample Control page.

FIG. **9** is part **1** of a sample Maintenance page, also referred to as a Maint page.

FIG. **10** is part **2** of the sample Maintenance page.

FIG. **11** is a sample Administration page, also referred to as a Admin page.

FIG. **12** is a partial sample of a Discover page.

FIG. **13** illustrates a three party model for online access to micro-server-enabled device documentation.

FIG. **14** is a simplified flowchart illustrating processing performed by a micro-server during micro-server initialization.

FIG. **15** is a simplified flowchart illustrating processing performed by a micro-server while getting run control from the device with which it is associated.

FIG. **16** is a simplified flowchart illustrating processing performed by a micro-server for executing a system thread.

FIG. **17** is a simplified flowchart illustrating TCP server thread processing by a micro-server.

FIG. **18** is a simplified flowchart illustrating HTTP server thread processing by a micro-server.

FIG. **19** is a simplified flowchart illustrating client subscription processing by a micro-server.

DETAILED DESCRIPTION

The operation of the micro-server is similar to that of a general-purpose web server. For example, FIG. **3** shows a remote thermostat device **300**, labeled Temp Sensor in FIG. **3**, equipped with a micro-server **302**, labeled μServer in FIG. **3**, connected to the same Ethernet network **304** as a client workstation **306**. The workstation could be used to monitor the remote temperature and/or to adjust the set point of the thermostat **300**. The remote device contains two software components: (1) the OEM code **308** that performs the actual control functions and (2) the software of micro-server **302**, which communicates with network clients. In this example, the workstation computer is considered a thin client, running no special software. All interactions with the remote micro-server-equipped thermostat device **300** is accomplished via a standard web browser program such as Netscape Communicator or Microsoft Internet Explorer. The micro-server-enabled thermostat device **300** could publish a standard web page containing the current temperature. Another page could be used to set the desired set point. Tremendous flexibility arises from the fact that multiple users, both local and far away, can access, view and change the information in the same manner without any special software, subject to configurable security measures.

Whenever a remote user accesses a micro-server-enabled device's web page, the micro-server software makes appropriate function calls to the OEM code to retrieve current information. Having done so, the micro-server incorporates the current information into the web page and sends the updated page to the requesting client machine. Similarly, whenever the user wishes to modify a control parameter, such as the set point of thermostat **300**, a user could enter the desired temperature into a text box and submit the new information to micro-server **302**. The micro-server **302** could then make appropriate function calls to the OEM code in order to change the setting.

In a first preferred embodiment of the invention, the micro-server is implemented in software. The micro-server could be provided to the OEM as a binary software library linked to the OEM application. Both the traditional static linking and the newer dynamic linking (DLL) methods are possible. In this embodiment, the OEM software could be linked with the micro-server libraries, which would then become an integral part of the micro-server-enabled device's embedded software. The micro-server software could then be placed in the non-volatile medium in the OEM's hardware, such as EPROM or EEPROM.

The arrangement of characteristics listed in Table 1, below, could be used. As will be apparent to those skilled in the art, other suitable arrangements could also be used without departing from the scope of this invention.

TABLE 1

Characteristics of the First Preferred Embodiment

Characteristic	Description
OEM software contained in	OEM hardware
micro-server software contained in	OEM hardware
TCP/IP stack contained in	OEM hardware
Ethernet Interface in	OEM hardware
OEM side of OEM-micro-server API	OEM code
micro-server side of OEM-micro-server API	micro-server code

In a second preferred embodiment, the micro-server could be an embedded-micro circuit board small enough to fit inside of the control box of a piece of industrial technology like a motor, a pump, a valve, or some other piece of process equipment. Of course, such an embedded-micro circuit board could also be placed into other types of devices and products, such as, consumer products. OEM software could be accommodated in the same board. In addition to the OEM and micro-server code, the software could include a TCP/IP protocol stack. As used in this specification and the appended claims, communication protocol dependent terms such as TCP/IP, HTTP, Ethernet, and the like, and means for supporting such protocols such as TCP/IP network protocol stack, and the like, are used for illustrative purposes only and should not be construed as limiting this invention to those particular protocols. As will be apparent to those skilled in the art, other appropriate communication protocols could also be used without departing from the scope of this invention. The hardware could include Ethernet support.

The characteristics of the second preferred embodiment of this invention could be arranged as in Table 2 below. As will be apparent to those skilled in the art, other suitable arrangements could also be used without departing from the scope of this invention.

TABLE 2

Characteristics of the Second Preferred Embodiment

Characteristic	Description
OEM software contained in	hardware controller card
micro-server software contained in	hardware controller card

TABLE 2-continued

Characteristics of the Second Preferred Embodiment

Characteristic	Description
TCP/IP stack contained in	hardware controller card
Ethernet Interface in	hardware controller card
OEM side of OEM-micro-server API	OEM code
micro-server side of OEM-micro-server API	micro-server code

In a third preferred embodiment, the micro-server could be a software component intended to run on a full WINDOWS platform (/95, /98, &/or /NT) or any other available operating system. This embodiment is useful in situations where the OEM's control platform is a computer capable of supporting a standard operating system environment. The operating system could provide TCP/IP support, and the computer's hardware could provide Ethernet connectivity. The characteristics of the third preferred embodiment of this invention could be arranged as in Table 3 below. As will be apparent to those skilled in the art, other suitable arrangements could also be used without departing from the scope of this invention.

TABLE 3

Characteristics of the Third Preferred Embodiment

Characteristic	Description
OEM software contained in	OEM computer
Micro-server software contained in	OEM computer
TCP/IP stack contained in	OEM operating system or add-on
Ethernet Interface in	OEM computer
OEM side of OEM-micro-server API	OEM code
Micro-server side of OEM-micro-server API	micro-server code

In a fourth preferred embodiment, the invention comprises hardware that does not contain the OEM code, the OEM code being embedded in its own hardware. The micro-server software is contained in its own micro-controller, with an Ethernet interface and an external hardware interface (e.g., serial, parallel, PC-104, etc.) to connect the micro-server's micro-controller to the OEM's micro-controller. In this embodiment, the OEM-to-micro-server API abstracts the hardware interface between the OEM hardware and the micro-server hardware. This embodiment is particularly useful for retrofitting existing devices for web operation.

The characteristics of the fourth preferred embodiment of this invention could be arranged as in Table 4 below. As will be apparent to those skilled in the art, other suitable arrangements could also be used without departing from the scope of this invention.

TABLE 4

Characteristics of the Fourth Preferred Embodiment

Characteristic	Description
OEM software contained in	OEM hardware
Micro-server software contained in	hardware controller card
TCP/IP stack contained in	hardware controller card
Ethernet Interface in	hardware controller card
OEM side of OEM-micro-server API	OEM code on OEM hardware
Micro-server side of OEM-Micro-server API	micro-server code on hardware controller card

A fifth preferred embodiment of this invention could use an embedded micro-controller built on a PC circuit board (ISA, PCI, etc.) that can be plugged into another computer. The micro-controller could contain the micro-server software and the TCP/IP network protocol stack. The OEM code could run on the computer that has the PC circuit board plugged into it.

TABLE 5

Characteristics of Fifth Embodiment

Characteristic	Description
OEM software contained in	OEM computer
micro-server software contained in	plug-in hardware card
TCP/IP stack contained in	plug-in hardware card
Ethernet Interface in	plug-in hardware card
OEM side of OEM-micro-server API	OEM code on OEM hardware
Micro-server side of OEM-micro-server API	micro-server code on plug-in hardware card

Each of the preferred embodiments comprises three Application Programming Interfaces ("APIs"): an OEM API, a TCP/IP stack API, and a System Services API. These APIs are suitable for use with current networking technology. It will be apparent to those skilled in the art that the concepts taught herein may be applied using API's other than those herein described and that the same or other API's may be used in connection with networking protocols developed in the future without departing from the scope of this invention.

The OEM API is used by the OEM software to configure the micro-server. The OEM API is also used to exchange data between the OEM software and micro-server software. The OEM API abstracts the micro-server to the OEM layer. The fundamental purpose of the OEM API is to provide a high level of abstraction for the micro-server from the OEM programmer's perspective. This allows the programmer to remain focused on the embedded application without having concern for the details of making the application web-enabled.

The OEM API is divided into five primary groups: (1) initialization group; (2) callback functions group; (3) system services group; (4) TCP/IP pass through group; and (5) scheduling group. An alphabetically organized listing of a micro-server OEM API functions including a synopsis of the call, the appropriate declaration, a narrative description of all arguments, the return value and associated usage notes are attached as Appendix A to this specification.

The OEM API Initialization group comprises a series of functions, which are called by the OEM software on power-up of the embedded device in order to inform the micro-server about variables which may be queried. For instance, a micro-server enabled servo-valve of the type typically used in chemical processing could be capable of providing data on its current setting (0–100%), fluid flow rate, and temperature. To make this data accessible to the outside world, the OEM program could advertise it to a co-embedded micro-server by executing a separate call to an appropriate API initialization function. Each such call could provide the name of the variable, e.g., Flow Rate, the units of measure, and a pointer to a callback function to be used by the micro-server to retrieve the value of that variable.

The OEM program could also execute similar API functions to inform the micro-server about its control points. The example servo-valve could have a single control point: the desired setting (0–100%). It could therefore make an appropriate call via the API to provide information about this control point and to provide a callback function that could be used to adjust this setting.

The OEM API callback functions advertised to the micro-server by the OEM software, could be used by the micro-server to retrieve the values of the OEM variables or to change device settings. The micro-server calling these callback functions will typically make up the bulk of interaction between the OEM software and the micro-server. Significantly, the OEM software is typically essentially unaware that the micro-server is making these calls. This is consistent with the desired abstraction of the micro-server from the embedded OEM software.

Preferably, the micro-server software is completely portable. Therefore, it typically does not make any assumptions regarding the presence or functionality of an underlying operating system. In order to provide the micro-server with very rudimentary system services, for example, access to non-volatile storage or timer services, the OEM software provides these to the micro-server, again, via callback functions. This occurs during initialization. As is similar to the data and control callback functions discussed above, the execution of these system callback functions is transparent to the OEM program.

In a standard micro-server-enabled device, the OEM software would not generally talk directly to TCP/IP clients. Such communications are typically handled by the micro-server and are invisible to the OEM software. The OEM API does, however, provide the ability for the OEM software to open a TCP server port and service special requests directly. When such requests arrive on the specified port, the micro-server passes them through to the OEM software. This occurs via a callback mechanism similar to those discussed above. The responses from the OEM software to the originating client are passed through the microserver in the opposite direction.

The scheduling group of the OEM API is comprised primarily of a function that is used by the OEM software to run the microserver software after the microserver has been initialized. The OEM software or hardware typically would make arrangements to allow the micro-server to run periodically. Since most embedded applications generally do not have an underlying operating system, the execution of the micro-server is typically controlled by either calling it periodically via a function from the OEM API scheduling group. Such a function call could occur, for example, by explicit periodic calls from the OEM software or by attaching the function to a timer interrupt.

In addition to the OEM API, each preferred embodiment comprises a second API, the TCP/IP stack API, which is used by the micro-server to communicate with the underlying TCP/IP stack. The TCP/IP API could be standardized to conform to the Winsock 1.1 interface standard. As will be apparent to those skilled in the art, other suitable interface standards may also be used without departing from the scope of this invention. The TCP/IP API abstracts the TCP/IP protocol stack to the micro-server.

Each preferred embodiment also comprises a third API, the System Services API, which is used to provide the limited service required by the micro-server. Since the micro-server is preferably software platform independent, the System Services API may be defined by the OEM code via the OEM API. As a consequence, the System Services API typically will not be usable by the micro-server until the initialization is complete. The System Services API provides the system services to the micro-server, while abstracting from the micro-server the details of providing such services.

The system services used by the micro-server during operation may be provided either by an underlying operating system or by the OEM code itself. The entity providing the services is typically transparent to the micro-server.

FIG. **4** is a simplified block diagram showing possible components of a micro-server. The OEM API is depicted by bold double-ended arrow **400**. The System Services API is depicted by bold double-ended arrow **402**. The TCP/IP API is depicted by bold double-ended arrow **404**.

System services **406** provides the system services required by the micro-server. This software is abstracted from the micro-server's point of view and may be either a part of the native operating system or part of the OEM software layer. Embedded OEM Software layer **408** communicates with device-specific hardware, except for the Ethernet hardware interface. Device hardware **410** may include sensors and other inputs, effectors or other outputs, and the like. The Ethernet hardware interface could be included in this layer, but is preferably accessible only from the TCP/IP stack. The main body **412** of the micro-server could comprise: micro-server-OEM API **414**; micro-server-TCP/IP API **416**; synthesized HTML **418**; OEM data image **420**; HTTP server **422**; and TCP/IP based server **424**.

Micro-server-OEM API **414** is an interface that could be used primarily by the OEM layer **408** to configure the micro-server and by the micro-server to exchange data with the OEM layer. The TCP/IP protocol stack could be used for communications over an Ethernet. TCP/IP protocol stack as used in this specification and the appended claims is not intended to be limited by the TCP/IP protocol. Rather, TCP/IP protocol is merely illustrative, and not intended to limit the scope of this invention to TCP/IP protocol. Other suitable protocols could be used without departing from the scope of this invention. Micro-server-TCP/IP API **416** is used by the micro-server to access the TCP/IP protocol stack **426**.

Synthesized HTML **418** can contain a dynamic copy of the HTML version of the web pages served by HTTP server **422** to a client. OEM data image **420** can cache the OEM layer data in order to minimize callback requests from the micro-server. Stale OEM data can be automatically refreshed. HTTP protocol server **422**, also referred to as HTTP server, can be used by the micro-server satisfy interactive HTTP request typically from thin browser-based clients. TCP/IP Based Server **424** could be used by the micro-server to satisfy TCP/IP based requests from clients executing default applets, manufacturer-supplied applets, client-side API software, or other TCP/IP-based software.

Each of the preferred embodiments is capable of five primary operating modes, as shown in the following table:

TABLE 6

Micro-Server Operating Modes

Mode	Operating Mode	Client Software
1	HTTP Server	Standard interactive browser.
2	HTTP Server with Client-side Refresh	Standard interactive browser
3	TCP/IP Server	Default applet, OEM-provided applet, Client-side API, Custom client
4	TCP/IP Server with Server-side Push	Default applet, OEM-provided applet, Client-side API, Custom client
5	TCP/IP Pass-through	OEM-provided applet

FIG. **5** depicts several possible server/client types of combinations according to the principles of this invention. The five micro-server operating modes are depicted by the boxes on the left side of FIG. **5**. They are: HTTP server **500**; HTTP server with client-side refresh **502**; TCP/IP Server **504**; TCP/IP Server with server-side push **506**; and TCP/IP pass through mode OEM server **508**. The five operating modes function as follows.

HTTP server **500** is the most common mode used in interactive access to a micro-server-equipped device from a thin-client workstation running a web browser. When a user lands on a desired device's default page, an appropriate HTML description of the current state of the device is dynamically configured by the micro-server and the resulting page is displayed on the client work station's screen. This information is typically not updated dynamically, and the user typically would click on the browser's RELOAD button to cause an update to occur. As depicted in FIG. **5**, HTTP server typically operates in conjunction with a standard browser **510**. A standard browser could include, but is not limited to, Netscape Communicator or Microsoft Internet Explorer. Either of these products, as well as many others both commercially available and custom designed, are capable of providing a sufficiently interactive client interface to access micro-server pages.

HTTP server with a client-side refresh **502** is similar to the HTTP server mode **500** except that the page displayed in the browser is continually updated at fixed time intervals. HTTP server with client-side refresh will typically be available if the micro-server-equipped device has been appropriately configured by the user. This mode also typically depends on a client-side refresh configured into the HTML description of the page being viewed. HTTP server with client-side refresh **502** typically operates in conjunction with a standard browser **510**, as shown in FIG. **5**.

The TCP/IP server mode **504** is non-interactive. Data is exchanged between a client program on the remote workstation and a TCP/IP server within the micro server. As shown in FIG. **5**, client applications which typically use this mode include: (1) a default client-side micro-server applet which determines the micro-server-equipped device's configuration and configures itself **512**; (2) an OEM-provided applet which adheres to the standard micro-server TCP/IP application packet format **514**; or (3) a micro-server client-side API application **516**.

A default applet **512** is a software entity provided by an applet server. Once acquired by a client, it becomes a part of the client browser and it will typically be capable of displaying device data in a convenient, graphical manner. The default applet could be written according to a standard micro-server application-to-application protocol definition and could communicate directly with the TCP/IP server within a micro-server. Since this communication takes place within the micro-server, underneath the OEM API, which provides networking abstraction, it is typically invisible to the OEM layer.

An OEM-provided applet **514** is functionally equivalent to the default applet, but may provide a richer set of functions. It could be acquired from the OEM either directly or via an applet procurement agent. The OEM-provided applet could communicate with the micro-server in different modes. If written to the standard micro-server application-to-application protocol definition, it communicates directly with the TCP/IP server within the micro-server, just like a default applet discussed above. The OEM could also write the applet to use a different application-to-application protocol. In this case, the applet would communicate with the OEM layer via the TCP/IP pass through mode **508**. This method would typically be discouraged because, under such circumstances, the micro-server would no longer provide full networking abstraction to the OEM software.

The Client-side API **516** is a software entity that can execute on the client machine and provide programmatic access to micro-server functions via an API. The Client-side API could use the TCP/IP server within the micro-server. In addition to handling ad-hoc requests from client applications, it is sufficiently intelligent to listen for broadcast messages from micro-servers to which it subscribes.

TCP/IP server with server-side push **506** is a mode similar to the standard TCP/IP server mode **504** but includes a limited push or broadcast capability to automatically initiate update data transfers to a limited number of selected clients. Data transfer destinations as well as trigger issues that initiate them may be user configurable. Judicious use of this option can result in significant savings in network traffic. As shown in FIG. **5**, TCP/IP server with server-side push typically operates with the same types of client-side entities as the standard TCP/IP server mode.

TCP/IP pass through **508** is a mode intended for special cases outside of the scope of normal micro-server operation. A specific example of the use of this mode is a scenario in which an OEM supplied applet **518** is running under the client-side browser, communicating with a specialized TCP/IP server in the OEM application itself. In this scenario, the micro-server simply passes all received TCP/IP packets to the OEM application via a prearranged call back function and replies to the originating client with packets provided by the OEM software. This functionality is supported by the micro-server for completeness and is typically not preferred because it does not take advantage of the network abstraction provided by this invention. Similarly, custom applications could be written that would execute on the client and make use of the TCP/IP server within the micro-server. Although such a use of TCP/IP pass-through mode is possible, it is not preferred because it does not take advantage of the network abstraction provided by this invention.

FIG. **6** illustrates that communications between the micro-server-enabled device and client-side software typically occur over three paths: (1) HTTP-protocol based communications between the HTTP server and the browser client, as shown at **600**; (2) TCP/IP based communications between client-side default or OEM-supplied applets and the TCP/IP based server, as shown at **602**; or (3) TCP/IP pass-through communications between the special client-side applications and the OEM software, with the micro-server acting as a pass-through intermediary, as shown at **604**. While paths **600**, **602**, and **604** could all be used simultaneously by one or more clients, typically path **600** will be used most often, while paths **602** and **603** will be used less often than path **600**.

A micro-server-enabled device could have five built-in web pages. These pages could all exist at the top level of the micro-server web site, so that accessing each would be easy. If no opening page is specified when a web server is browsed to, the web server opens a page called default.htm, also referred to as the site's home page. Another way to access the home page could by adding/read to the end of the URL. The home page for a micro-server-enabled device could publish the device's parametric data.

The control page is available by adding/control to the end of the URL. This page provides authorized persons access to the control functions, if any exist, on the device. For example, a servo-valve could be outfitted with the ability to set its flow rate from 0 to 100%.

The maintenance page is available by adding/maint to the end of the URL. This page could provide access to the maintenance functions and/or the maintenance record of the device.

The administration page is available by adding/admin to the end of the URL. This page is used by authorized persons to set operating characteristics of the micro-server such as the physical location name, the out-of-limits parameters for voltage, current, flows, and temperatures, push IP addresses, and the like. In general, it is used to tailor the device to its environment.

The discover page can be accessed by adding/discover to end of the URL. This page supplies information to clients during automatic discovery mode.

In order to clarify how the micro-server pages are accessed, assume that a unit has been configured with an IP address of 202.133.3.4. In addition, a local DNS server has mapped the same IP address to a URL of ServoValve37. Table 7 below demonstrates how the four pages are accessed either with or without the DNS server:

TABLE 7

Accessing micro-server Pages with or without DNS

Micro-server Page	With DNS	Without DNS
Home (Default)	http://ServoValve37	202.133.3.4/default
Control	http://ServoValve37/control	202.133.3.4/control
Discover	http://ServoValve37/discover	202.133.3.4/discover
Administer	http://ServoValve37/admin	202.133.3.4/admin
Maintenance	http://ServoValve37/maint	202.133.3.4/maint

In order to simplify navigation, micro-server could automatically provide each page with links to the other four pages. FIG. **7** shows a typical micro-server page, in this case, a default page for a device controlling a Servo-valve. The screen is divided into two frames, the upper reflecting mostly static information and the lower containing the device read data. Buttons located on the left-hand side in the upper frame lead to the other four micro-server pages. As can be seen, the micro-server presentation can be very much like that of a standard page on the world-wide-web. This choice of presentation allows the use of client-side software, in this case a standard Netscape browser, which is familiar to most people. As will be apparent to those skilled in the art, the web pages presented in FIGS. **7–12** are representative of one possible implementation and other suitable implementations are also possible without departing from the scope of this invention.

In FIG. **7**, the Auto Refresh feature is depicted as turned off in the upper right hand corner of the sample control page. Therefore, the page, as displayed in the browser, would remain static even though the device data may have changed. If the Auto Refresh feature were on, the page would be updated periodically. Underneath the Auto Refresh indication, the page contains both the current micro-server device time as well as the time of the last update. The latter refers to the time when the last set of data readings was obtained by the micro-server from the OEM layer. All data values are typically updated simultaneously.

The following fields shown in FIG. **7** would typically have been configured by the administrator, either via the Admin page or via the Client-side API: User Description; User Id; User Location; and Nominal, Low, and High Data Values for all device parameters. Once configured, these fields not only personalize the device but also change the way the micro-server reports data.

All device parameters are tagged with a unique identifier as depicted in the Id. column of the home page depicted in FIG. **7**. Although not very important in the interactive presentation depicted in FIG. **7**, this is a significant feature of the micro-server according to this invention. It allows parameter level access to micro-server data via the Client-side API. In addition, and also of significance, is the ability to access individual device parameters in other HTML constructs, allowing construction of alternate and/or hierarchical device and system views.

FIG. **8** depicts a sample control page. Any modifications of operating parameters could be subject to a security restriction imposed by a system administrator.

FIGS. **9** and **10** depict parts **1** and **2**, respectively, of a sample maintenance page. The maintenance page could provide a very useful maintenance log that could be stored in non-volatile memory. If non-volatile memory is not available in the micro-server-enabled device, the maintenance log could be kept in a local maintenance server. The manufacturer's micro-server license Id. and the device's serial number, both of which could be available as micro-server variables, could form a unique identifier to be used in tracking the maintenance functions of the device.

FIG. **11** depicts a sample Admin page. FIG. **12** depicts a sample Discover page. The contents of the Discover page can be used by any authorized client to establish detailed understanding of a newly discovered micro-server-enabled device, for instance, a micro-server enabled device recently coupled to a network. Typically, micro-server variables uniquely identify important pieces of information related to the characteristics and the operation of the micro-server-enabled device. Micro-server variables may be used by the application-to-application protocol supported by the TCP/IP based server within the micro-server. Micro-server variables would typically be involved with the operation of the client-side API.

Web pages are expressed in HTML (HyperText Modeling Language) before they can be displayed by a browser. When a client browser lands on a given page, it requests the HTML description of the page to be sent. The HTTP server responds by sending the HTML and the browser, having received it, displays it on the client computer's screen. Upon receiving a request for HTML from a client, the micro-server makes appropriate calls to the OEM layer in order to retrieve appropriate data, constructs an HTML description of the requested page, and then sends the page to the client.

To reduce the micro-server's overhead of frequent calls to the OEM layer for retrieving data updates, the micro-server can cache OEM data. The cache may include a date/time stamp of the last update. Upon receiving an HTTP request from a browser client, the micro-server can check whether the data is available in the cache (it usually is, except for during startup) and then checks the cache timestamp. If the data is reasonably recent, no update is performed and the last version of the dynamic HTML is sent. If the data is stale, however, the micro-server can update it by performing the required callbacks to the OEM layer, update the cache and the associated timestamp, re-construct the HTML, and send it to the requesting client.

The maximum data latency for HTTP clients can be specified via the OEM API by calling a function defined by the OEM API. The maximum data latency is the maximum acceptable age of the data cached by the micro-server, as served to HTTP clients. It would typically be the responsibility of the OEM to make reasonable estimates for this parameter in order to reduce the overhead of frequent updates and re-construction of the HTML page description. For example, if the parameter measured by the micro-server-equipped device is temperature and the sensor has a limited frequency response, specifying a short latency time would be counterproductive because frequent data updates would not be necessary and would consume excessive resources.

Maximum data latency for TCP/IP clients is controlled via a different OEM API function call. This parameter is typically smaller than the maximum data latency for HTTP clients since TCP/IP clients are usually higher in performance and not subject to the delays of the human-machine interface such as in an interactive HTTP-based browser.

From the client perspective, a micro-server appears to be a normal, fully functional HTTP server executing on a conventional computer with a fully functional file system. The reality, however, could be quite different. A typical hardware platform could comprise an embedded processor without a real file system. Furthermore, the amount of non-volatile memory available to a micro-server could be quite small. In order to maintain full HTTP compliance, the micro-server, typically abstracts or hides a number of important details from the client, maintaining full browser compatibility.

In the preferred embodiments, by default, the micro-server will not broadcast any data unless the device has at least one registered subscriber and at least one trigger condition has been specified.

The configuration of server-side push could be performed interactively via the browser interface and the Admin page or could be performed via the TCP/IP application-to-application protocol. In either case, it could be subject to security checks. A subscribing client may choose to unsubscribe from broadcasts from any micro-server-equipped device by removing its host name or IP address from the subscription list. Such un-subscription requests could be subject to normal security measures. The broadcast data could be sent to the IP address of the subscribing host or hosts and a specified port, such as TCP/IP port **162**. Client-side software entities which have registered their subscription with any micro-server-equipped devices would typically have to be listening to this port in order to receive the broadcasts.

In the pass-through TCP/IP mode, an OEM-supplied applet could be running under the client-side browser, communicating with a specialized TCP/IP server in the OEM application itself. In this scenario, the micro-server could pass all received TCP/IP packets to the OEM application via a prearranged callback function and could reply to the originating client with packets provided by the OEM software.

The OEM layer could use the OEM API to request the micro-server to set up multiple pass-through connections of this type, each with an arbitrary associated TCP/IP port number. The choice of the port number should not conflict with standard port numbers used by the micro-server or any other well known port numbers as defined in standard TCP/IP RFCs. The responsibility to use correct port numbers would typically lie with the OEM layer.

The micro-server is typically initialized each time power is applied to the device into which it is embedded or to which it is associated. The sequence of events could proceed as follows: (1) power is applied to the device; (2) the OEM layer is bootstrapped; (3) the OEM layer performs various initializations (other than initialization of the micro-server); (4) the OEM layer retrieves the IP address of the device from appropriate hardware such as a dip switch, or the like; (5) the OEM layer begins the micro-server initialization; (6) the OEM layer performs the micro-server initialization via the OEM API, passing the micro-server the IP address; (7) the OEM layer terminates micro-server initialization; (8) the OEM layer makes arrangements to start the micro-server execution thread; and (9) the OEM layer initiates its normal execution. With respect to step (4), alternately, a dynamic IP address assignment could be obtained from a local DHCP server.

The micro-server is typically not operational until the initialization has completed and an execution thread has been arranged for it, at which point, it typically starts operating normally.

Auto-discovery enables new micro-server-enabled devices to become useful immediately after plugging them into a network. Auto-discovery typically occurs in two separate steps: (1) new devices are automatically discovered by an auto-discovery and view server that scans the network for new devices. The scans can be configured to occur periodically or can be run on-demand. Once discovered, a new device can be automatically inserted into the main view on the server and can be accessed simply by clicking on its name. This operation could cause the client browser to be transferred to the device's home page; and (2) an automatic attempt can be made to load an OEM provided viewing/control applet to the client computer. If this operation succeeds, the applet can become operational immediately. If the attempt fails, however, a default applet can be loaded from the Local Applet Server. The applet can be initiated, causing it to perform the second, detailed discovery of the node itself. The TCP/IP application-to-application protocol is used to glean from the node, information about its characteristics, variables, recommended settings, etc. The same information could be presented in human-readable form on the device's discover page.

An OEM could configure the micro-server to include standard HTML links to relevant device documentation located on an OEM web site or some other web site associated with the device. This configuration of the micro-server could take the form of a simple call to the micro-server via the OEM API.

This call could result in a Documentation link created on the device's Maintenance page. Immediate and seamless access to on-line documentation, including drawings, specifications, parts lists, and manuals, of any micro-server-equipped device could be provided simply by clicking on the Documentation link on the device's maintenance page from any client browser. FIG. **13** illustrates how this could work.

In the preferred embodiments, access to online device documentation located on the OEM's web site could be a four-step process, as illustrated in FIG. **13**. First, the user could click on a Documentation link created on the device's Maintenance page, as indicated by arrow **1300**. Second, the URL of the documentation could be sent back to the client browser by the HTTP server component of the microserver; as shown by arrow **1302**. Third, the browser could request the documentation, as specified by the URL, from the OEM's website, as depicted by arrow **1304**. Fourth, the browser could load the requested documentation from the manufacturer's site as shown by arrow **1306**. Advantageously, although the online documentation exists on the manufacturer's web site, the hyperlink provided on the micro-server maintenance page makes it appear that the documentation actually resides in the device itself.

The operation of the micro-server could be language independent. Object files containing the fixed string content for various languages could be provided. Parameter description strings, maintenance log entry strings, etc., as passed to the micro-server via the OEM interface, could be language independent. In addition, a country code could be supported for configuring certain internationalization features, including, but not limited to, regional seasonal time shifts. In an appropriately configured micro-server, simultaneous multilingual operation is possible. Similarly, a micro-server could be configured to report time to each client in each client's local time.

Time could be maintained as Universal Coordinated Time (UTC). The time information could be obtained via an appropriate API interface advertised to the micro-server via the OEM interface. Although the notion of UTC may not necessarily be needed in a given application, it could be extremely useful when coordinating real time information from multiple sources.

If an IP address or a hostname of a local time server has been specified, the micro-server could attempt to obtain local time information directly and adjust its clock accordingly. If a local time server is not available and the country code has not been set, the micro-server could be configured with the values of the time difference from UTC and any local/seasonal time offset. Regardless of how the date and time are originally set, they could continue to be available as both UTC and local values.

The micro-server's notion of time could based on local time obtained from system services and corrected with a local offset in hours from UTC contained in a variable TZ1. Application of the offset could provide the local time. The notion of daylight-savings time as well as any special regionally-based time offsets could be handled by a time internationalization function within the micro-server, resulting in an appropriate value of a variable, TZ2. This function could be used in the event that a timeserver could not be accessed on the subnet. The local time could be obtained by summing the UTC with TZ1 and TZ2, as follows:

$$T_{local}=T_{UTC}+TZ1+TZ2$$

If a local time server is not available and the country code has not been set, the micro-server could be configured with the values of the time difference from UTC and any local/seasonal time offset (variables TZ1 and TZ2).

The micro-server could use hostnames, IP addresses, and URLs, in different contexts. IP addresses are, effectively, hardware addresses of computers communicating over Ethernet. The micro-server could use IP addresses to specify addresses of clients subscribing to automatic data updates (Server-side push), the address of a local Timer Server, and the like.

In all cases where a machine has a specific name, for example, Bldg45_host, that name could be used in configuring the micro-server. This approach could provide added flexibility, since the IP addresses of a given machine can be changed without the need to re-configure micro-server-equipped devices which reference it. To accomplish this, the micro-server could make use of a DNS (Domain Name Service) in order to look-up the IP address of a host. This service could be provided by a DNS server connected to the same network as multiple micro-servers. In order to use the DNS service, micro-servers could be configured with an IP address of a local DNS server.

URLs, or Uniform Resource Locators, are typically addresses of a specific file or another Internet resource. They are typically formed by adding a file pathname to the address of the device. The hostname or an IP address can be used as a base-name for the path, for example VTG_applets/servovalves/GWLinc/model3500, or 124.67.98.1/servovalves/GWLinc/model3500. URLs could be used by a micro-server to point to items such as manufacturer documentation for a specific device, an Applet used to view the device's data, and the like.

A micro-server could support Parameter level URL (PURL). A PURL can be used to reference a particular parameter published by a micro-server-equipped device. For example, the PURL slurrypump3/da0013.dat,value could refer to the value of data parameter **13** on micro-server called slurrypump3. Similarly, 128.45.33.11/mfgname.dat could return a string containing the manufacturer name of the micro-server-equipped device at IP address 128.45.33.11. The qualifier which follows the comma (,) separator could be an additional part of a request to specify an attribute of the parameter to be retrieved.

The operation of micro-server-enabled devices could be augmented by several additional software components such as: Default Applet Server; Time Server; Maintenance Server; Auto-discovery and View Server; Applet Procurement Server; and Local Applet Server/Procurement Agent.

While applets are not necessary for accessing micro-server-equipped devices with a browser, the Default Applet Server could provide default applets for interfacing with micro-server devices from within a standard client browser. The default applets could be used in situations where either (1) the manufacturer did not provide a specific viewing/control applet, or (2) the applet cannot be reached because of insufficient network connectivity. The server could contain standard default applets and distribute them, if necessary, to the browser clients. This operation could be handled by an applet procurement script embedded into the HTML code loaded from the micro-server-enabled devices.

The Time Server could provide the current UTC time to the requesting micro-server nodes, upon request. In addition to the UTC time, it could provide the hour offset from UTC and a second local time offset. Both offsets could be in hours. The Time Server could present a standard micro-server interface, in which case its services could be accessed by any software client via the standard client-side API.

The Maintenance Server could be a specialized entity used to provide maintenance logs for those micro-server-enabled devices that have a very limited amount of, or no, non-volatile storage. The server could essentially form an external repository for maintenance data hyper-linked to the device, so that the maintenance server's operation would be essentially transparent to the client.

The Auto-discovery and View Server could be a focal point of user access to a web of micro-server-enabled devices. The Auto-discovery and View Server could automatically discover micro-server devices within a given range of IP addresses. It could be run on demand or configured to run periodically to update the mappings. Auto-discovery could allow new micro-server enabled devices to publish device data simply by plugging the device into a network. The Auto-discovery and View Server could generate a local HTML master index page (Main View) of all devices, usable by a browser. This list could become a focal point for many users. Individual access points (URLs) could, of course, be book-marked on a user's browser in a manner well known in the art.

The Auto-discovery and View Server could generate a local database file (text, Access, etc.) with all device mappings, useable by other programs (applets, applications, etc). This list could be useful for configuring devices remotely.

The Auto-Discovery and View Server could allow the system administrator to configure system-wide monitoring and control panels that could present different views of any number of devices in the system. Each data point, or any other micro-server variable, could be monitored individually by referring to the variable's PURL. Once configured, views could be served-up to other clients and the data could be monitored live via the established views. Typically, an OEM applet would be available either locally or from the OEM server and would interact with the Auto-discovery and View Server.

The Local Applet Server could locate and retrieve device-specific applets from an OEM's server and make them available on a local facility server for access by clients. If direct Internet access is limited due to restricted connectivity, the Local Applet Server could attempt to procure the correct applets from the manufacturer via electronic mail. Once the applets are procured, they could be cached on the Local Applet Server. If the applets cannot be procured, and before they can be successfully procured, the Local Applet Server could use a substitute default applet. Once loaded on the client machine, the default applet could perform detailed discovery of the device. References to these client applets could occur on the micro-server pages that are served up. Since the applets are usually cached on the client machines, the overhead required to load them would typically be experienced only on initial use.

There are several types of client software which could interact with the micro-server. They are Standard Browser; Client-side API; Standard Applet; OEM-provided applet; and Custom Software.

Any standard browser (Netscape, Internet Explorer, etc.) running on any workstation platform (WINDOWS) 95/NT, UNIX, etc.) can be used to interact with micro-server-enabled devices or any of the servers listed above. OEM-provided viewing/control applets could be automatically loaded if desired. The location of a thin browser is essentially irrelevant. It could be connected to the same sub-net as the devices or it could access the network remotely. The workstation used to interact with a device can be chosen ad-hoc, since no special software is needed.

The purpose of the client-side API is to provide program access to the micro-server-enabled devices and other servers. It could allow custom monitoring and control applications to be developed and executed on the client, with full access to the device and server data. The client-side API could provide multi-threaded access to micro-server-enabled devices and servers. It could be used by standard OEM-provided viewing/control applets as well as by the default applets. The client-side API calls could allow reading and writing of individual variables from/to a specified device, subject to configurable security restrictions, and it could perform auto-discovery of devices that it communicates with. The client-side API code could perform local caching of device data. This would reduce network traffic since some of the program data requests, depending on requested priority, could be satisfied locally.

OEM-provided applets could be interactive viewing/control software modules that could be automatically loaded into the client browser from the OEM web site. This operation can be totally transparent to the client.

OEM-provided applets could be matched to the micro-server-enabled eliminating any need to perform detailed discovery. Such applets could check and validate the device OEM release level and disable themselves if they were no longer current. The applet procurement mechanism could then be re-activated, repeating the above cycle.

Default applets could be general purpose in nature, perhaps lacking some of the features of OEM applets. They would typically be uniform in appearance and devoid of the personalization of OEM applets. Default applets could be obtained from the local applet server.

Custom software developed for the client could use the client-side API to communicate with micro-server-enabled devices or servers.

The micro-server software is comprised of multiple tasks, referred to as threads. The individual threads are preferably maintained by the micro-server software itself. The micro-server software typically does not block, since that would likely interfere with operation of the OEM code which is responsible for operation of the micro-server-equipped device itself. The micro-server software typically gains processor control periodically and preferably relinquishes such control as soon as is reasonably possible.

The micro-server could run only when it is called at one of its standard entry points. The initialization API calls could account for the majority of these and could be used to configure the micro-server software based upon the device in which the micro-server software is embedded or with which it is associated. The OEM code could make these calls upon each bootstrap.

Referring to FIG. **14**, processing is depicted as being initiated by the OEM software layer at **1400**. If the OEM software layer has called us_InitStart(), the software begins the initialization process, as shown at **1402**, **1404**, and **1420**. If us_InitStart() has not been called, any other initialization function calls are simply ignored, as indicated at **1406**, **1408**, and **1410**.

The OEM code calls us_InitDone() to signal the end of initialization, triggering validation of the initialization sequence, as shown at either **1412**, **1414**, **1416**, and **1410** or **1412**, **1414**, **1416**, **1418** and **1420**. When the OEM layer calls us_InitDone(), the micro-server could validate the entire initialization sequence as shown at **1414**. If the initialization sequence was successful, the TCP/IP protocol stack can be initialized to enable communications, as shown at **1416** and **1418**. The TCP/IP stack could be responsible for the initialization of the underlying TCP/IP hardware, the detailed workings of which are abstracted from the OEM layer by the micro-server. A call to us InitStart() could clear all of the micro-server data structures and could be the first step of the initialization process. Any OEM call with inappropriate arguments could cause a failure.

If the call was neither us_InitStart() nor us_InitDone(), then it could be processed as either an initialization call or a TCP passthrough request being made by the OEM code, as shown at **1406**, **1412**, and **1422**. TCP passthrough requests could cause the micro-server to blindly pass TCP/IP requests through. The validity of the call could be checked. Arguments and context could be validated for both initialization and TCP passthrough calls. If all is well, the return code could inform the OEM accordingly, as shown at **1424** and **1420**. Similarly, any errors could result in an appropriate error code being returned, as shown at **1424**, **1408**, and **1410**.

Referring to FIG. **15**, the micro-server could gain control via an explicit call from external software, as shown at **1500**. This could occur via an explicit call to the function us_Run() made periodically from the OEM software or from an interrupt service routine tied to a timer. This is the principal entry point for post-initialization processing by the micro-server. Once entered, the micro-server could execute a system thread, as depicted at **1502**, and one thread associated with an active HTTP connection, if any exist. The details of the system thread are presented in FIG. **16**. The entire execution point is preferably non-blocking in order to return quickly to the mainstream OEM software.

If there are no active HTTP connections, processor control can be returned to the OEM software, as depicted at **1504** and **1506**. If there are active HTTP connections, the next HTTP connection thread to be executed can be selected from a process table, as depicted at **1508**. Since there is at least one active HTTP connection thread, an HTTP connection thread can be executed, as depicted at **1510**. The details of executing an HTTP Connection thread are presented in FIG. **18**.

If the HTTP connection thread execution has completed, the HTTP connection thread can be removed from the process table and processor control can be returned to the OEM software, as depicted at **1512**, **1514**, and **1506**. If the HTTP connection thread has not yet completed, for instance, because it is waiting for an external event, the HTTP's connection thread execution state can be retained in the process table.

Referring to FIG. **16**, the System thread can be entered each time the micro-server software executes after initialization has been successfully completed. The system thread could perform all micro-server functions except handling HTTP connections. The System Thread could initially check for any HTTP requests on the TCP socket bound to port **80**. Port **80** is the standard port for receiving HTTP requests originated by client browsers. If there is a pending HTTP request, a new HTTP connection thread could be created and scheduled as the next HTTP connection thread to be executed, as depicted at **1600**, **1602**, and **1604**. New HTTP connections are preferably scheduled as high priority since accepting the HTTP requests as soon as they arrive allows the micro-server to process them more quickly.

Requests on the TCP server port originate from client entities other than standard interactive browsers and could be handled differently, according to a traditional client-server model, as depicted at **1606**, **1608**, and **1610**. Such requests typically originate from client side applets or micro-server APIs. If there is a pending TCP client request, the TCP request could be processed and a response is sent to the originating client. Details of this process are presented in FIG. **17**.

If there is a pending TCP pass-through ("PT") request on the ports specified by the OEM software during initialization, the TCP pass-through request is processed and passed to the OEM software via a callback function specified during initialization, as depicted at **1612**, **1614**, and **1616**. Pass-through requests are literally passed through by the micro-server, in both directions, without interpretation.

If there are any subscribers, client subscriptions could be processed, as depicted at **1618** and **1620**. Details of subscription processing are presented in FIG. **19**. Subscribers are clients which have registered with the micro-server for automatic push notification upon the existence of certain triggers. As depicted at **1622**, processor control is then returned to the Get Run Control thread depicted in FIG. **15**.

Referring to FIG. **17**, the TCP server thread can process requests for data originating from non-browser clients. These clients typically are client-side applets or micro-server APIs. This thread can be entered from the System thread upon detection of the presence of such requests.

Initially, the TCP Server Thread could check whether OEM initialization of the micro-server has been successfully completed, as depicted at **1700**. If not, as depicted at **1702**, **1704**, and **1706**, an error packet could be returned to the originating client and control could be returned to the System Thread depicted in FIG. **16**. If the micro-server has been successfully completed, the priority could be extracted from the client request, as depicted at **1708**. The priority could be used to determine whether the data in the OEM data cache is sufficiently current to satisfy the request. The concept of request priorities and OEM data caching could be used to satisfy client requests with reasonably recent OEM data without having to get those data values from the OEM upon receiving each request. If the OEM data is stale (i.e., not sufficiently current), the cache could be refreshed by obtaining current OEM data via execution of callback functions specified during the initialization process, as depicted at **1710** and **1712**. The cache could be time-stamped each time this happens.

If the client is requesting the value of a single micro-server variable, a single-variable response packet could be generated, as depicted at **1714** and **1716**. If not, a multi-variable response packet could be generated, as depicted at **1718**. As depicted at **1704**, and **1706**, the response packet could be sent to the originating client and control could be returned to the micro-server System Thread as depicted in FIG. **16**.

Referring to FIG. **18**, the HTTP server thread could be executed for each incoming HTTP request originated by a client browser or another HTTP-compliant entity. The HTTP server thread could be called from the micro-server Get Run Control thread depicted in FIG. **15**. Only one HTTP connection thread is typically executed during a single invocation of the micro-server from the OEM software, with each subsequent execution processing the next active HTTP thread. The HTTP request could be retrieved from the TCP socket bound to port **80**, as depicted at **1800**. If the HTTP request is a valid GET request, the data cache could be checked for staleness, the data cache could be updated, if appropriate, and the HTML defining the requested document could be regenerated, as depicted at **1802**, **1804**, and **1806**. Validation of a GET request could include validating the request format and the specified document. Using a standard file system paradigm, document pathnames could be resolved relative to internal micro-server data structures. Dynamic fields, such as the date, time, and an update counter could be updated, as shown at **1808**.

If the document pathname specifies attributes of a micro-server variable, an alternate HTML document could be generated specifying the requested variable information, as depicted at **1810** and **1812**. Otherwise, a standard response to the GET request could be generated including the requested document, as depicted at **1810** and **1814**.

If the client request is a valid HEAD request, the HTML header information for the requested document could be constructed, as depicted at **1816** and **1818**, and a complete HTTP response to the HEAD request could be generated, as depicted at **1820**. The validation of HEAD requests can include validating the request format and the specified document. Using the standard file system paradigm, document pathnames could be resolved relative to the internal micro-server data structures.

POST requests could be used to write data to the micro-server. If a valid POST request is received, the micro-server could internally simulate a CGI server to process the request, as depicted at **1822** and **1824**. If the data being written by the client affects OEM data, as opposed to exposed micro-server interface data, the OEM data cache could be updated and the affected OEM data values could be communicated to the OEM software via callback functions specified during initialization, as indicated at **1826**. The HTTP response could be generated to the POST request, as depicted at **1828**. The HTTP response could be sent to the originating HTTP client, as depicted at **1830**. As depicted at **1832**, control is then returned to the micro-server main run thread shown in FIG. **15**. In the event of an error, the micro-server could construct a standard HTML error response specifying the appropriate error number and description, as depicted at **1834**.

Referring to FIG. **19**, subscription processing could be entered from the micro-server System thread, as depicted at **1620**. Subscription processing's function is to send updates to subscribing clients. Initially, the next subscription could be retrieved from the subscription table, as depicted at **1900**. Subscribers could be identified by their IP address. If there are any remaining subscriptions to be processed, the subscription triggers could be retrieved, as depicted at **1902** and **1904**. Triggers are typically specified by subscribing clients. For example, a trigger could be based on an alarm on a specified data variable or an elapsed time period. If none of the triggers for the current subscriber have been satisfied, processing could proceed to the next subscriber, as depicted at **1906** and **1900**. The OEM data cache could be refreshed, if necessary, as depicted at **1908** and **1910**. The response to the client request could be generated and sent to the client's IP address, as depicted at **1912** and **1914**. As depicted at **1902** and **1916**, once all subscriptions have been processed, control could be returned to the micro-server system thread, which is depicted in FIG. **16**.

APPENDIX A—Micro-Server OEM API Definition

The information contained in this section is an alphabetically organized listing of the micro-server OEM API. This API forms an interface between the OEM software layer and the micro-server itself. The API consists of several functional groups. They are:

Micro-server initialization with constant information

Micro-server initialization with initial operating parameters

Micro-server initialization with system services data

Micro-server initialization with device read/write data

For each of the functions comprising the OEM API, there is a synopsis of the call including the appropriate declaration, a narrative description, description of arguments, the return value, and associated usage notes.

us_DeleteMaintLog()

Micro-Server Operation

Description

Deletes a message entry from the device maintenance log.

Synopsis

#include <userver.h>

int us_DeleteMaintLog(int message_id)

Arguments

The following arguments are passed to this function:

message_id message identifier obtained from a previous call to us_WriteMaintLog().

Return Value

This function returns an identifier upon success and −1 upon failure. In the event of failure, the external variable Us_error contains an appropriate error code. Error codes are defined in userver.h.

Usage Notes

This function is called to delete a text entry from the device non-volatile maintenance log. The entry would have been previously made by calling us WriteMaintLog(). If non-volatile memory is not present in the Micro-server-enabled device and the Micro-server has been remotely configured with a name or IP address of a local maintenance server, the deletion is made from the server log via an appropriate TCP/IP transaction.

See Also

us_WriteMaintLog()

us_GetStatus()

Micro-Server Operation

Description

Returns current Micro-server status

Synopsis

#include <userver.h>

int us_GetStatus(void)

Arguments

none

Return Value

This function returns a 16-bit bitmask representing logically ORed Micro-server status. Normal operating status is 0. Individual bit flags are defined in userver.h.

Usage Notes

This call can be made in any context.

See Also

us_Run()

us_GetVariable()

Micro-Server Operation

Description

Returns a value of a variable

Synopsis

#include <userver.h>

char *us_GetVariable(char *var_name)

Arguments

The following arguments are passed to this function:

var name Character string containing the name of the variable

Return Value

This function returns a pointer to the value of the requested variable. The pointer should be immediately used since it will become invalid with a next call to any API function. If the variable name passed to the function is invalid, this function returns a NULL pointer.

Usage Notes

This function is primarily used to retrieve values of variables set by the client, and its use is discouraged in normal operation. Values of data variables cannot be retrieved via this mechanism, since their variable names are dynamically assigned by the Micro-server code during initialization and, in particular, since their values are already known to the OEM layer. Any attempt to retrieve data variable values by "synthesizing" their names will yield unpredictable results.

us_InitDone()

Micro-Server Initialization

Description

Concludes the Micro-server initialization process.

Synopsis

#include <userver.h>

int us_InitDone(void)

Arguments

None

Return Value

This function returns 0 (US_OK) if the initialization process has been performed correctly and –1 upon failure. In the event of failure, the external variable Us_error contains an appropriate error code. Error codes are defined in userver.h.

Usage Notes

Us_InitDone()should be the last of a set of calls performed to initialize the Micro-server. The server will not begin operation until the call is made with a value of US OK returned to the caller.

us_InitMem()

Micro-Server Initialization

Description

Initializes the Micro-server memory

Synopsis

#include <userver.h>

int us_InitMem(char *memory, unsigned long memsize, unsigned long diskspace, char *directory,)

Arguments

The following arguments are passed to this function:

memory Points to the first byte of a memory block to be used for Micro-server data space. The caller should insure that the address is word-aligned.

memsize Specifies the size of the data area available to Micro-server, expressed in bytes.

diskspace Specifies the amount of disk storage available for use by Micro-server. If the number is 0, Micro-server will not use disk I/O.

directory If the hardware environment in which Micro-server executes includes non-volatile disk storage, this argument should contain the path for the creation of Micro-server files, e.g. "c:\lus_files". If this argument is a null string, and diskspace is non-zero, Micro-server will create file(s) in the current directory.

Return Value

This function returns 0 (US_OK) upon success and –1 upon failure. In the event of failure, the external variable Us_error contains an appropriate error code. Error codes are defined in userver.h.

Usage Notes

If disk storage is not available in the execution environment, functions facilitated by non-volatile memory can still be used by making a call to us_Non Volatile().

See Also

us_NonVolatile()

us_Initstart()

Micro-Server Initialization

Description

Micro-server initialization process.

Synopsis

#include <userver.h>

int us_InitStart(void)

Arguments

None

Return Value

This function always returns 0 (US_OK)

Usage Notes

Us_InitStart() should be the first of a set of calls performed to initialize the Micro-server. If the call is made again, initialization starts all over.

us_Non Volatile()

Micro-Server Initialization

Description

Informs the Micro-server about non-volatile memory present in the execution environment.

Synopsis

#include <userver.h>

int us_NonVolatile(char memory, unsigned long memsize, void (*nvwrfunc)(), unsigned char (*nvrdfunc)())

Arguments

The following arguments are passed to this function:

nvmemory Points to the first byte of a non-volatile memory block to be used for Micro-server non-volatile data.

nvmemsize Specifies the size of the non-volatile data area available to Micro-server, expressed in bytes.

nvwrfunc Specifies the address to an OEM-supplied function, which returns a void, used to write a single byte of data to the nonvolatile data area. Micro-server assumes that the function is used as follows:

```
unsigned char written data;
char *address;
nvwrfunc(address, written_data)
```

nvrdfunc Specifies the address to an OEM-supplied function, which returns an unsigned char, used to read a single byte of data from the nonvolatile data area. Micro-server assumes that the function is used as follows:

```
unsigned char read_data;
char *address;
read_data=nvrdfunc(address)
```

Return Value

This function returns 0 (US_OK) upon success and –1 upon failure. In the event of failure, the external variable Us_error contains an appropriate error code. Error codes are defined in userver.h.

Usage Notes

The OEM program should call NonVolatile() if disk storage is not available in the execution environment. If neither disk storage or non-volatile memory are available, Micro-server functions which require non-volatile memory will either not be available or will not be non-volatile.

The user program should define the nvrdfunc() and nvwrfunc().

us_Run()

Micro-Server Operation

Description

Passes control the Micro-server

Synopsis

#include <userver.h>

int us_Run(void)

Arguments

none

Return Value

This function returns a 16-bit bitmask representing logically ORed Micro-server status. Normal operating status is 0. Individual bit flags are defined in userver.h.

Usage Notes

The purpose of us_Run() is to provide execution cycles to the Micro-server. This is the fundamental mechanism, which is used to operate the Micro-server. Under normal circumstances, the call to this function is made from within a loop in the OEM software layer. Alternately, the call can be made from an interrupt service routine for the real-time clock. The call is non-blocking, since the Micro-server adheres to"passthrough" architecture with no loops. Execution occurs on the stack in the current context.

See Also

us_GetStatus()

us_SetHttpLatency()

Micro-Server Initialization

Description

Sets up the Micro-server HTTP latency.

Synopsis

#include <userver.h>

int us_SetHttpLatency(long latency)

Arguments

The following arguments are passed to this function:

latency HTTP latency in milliseconds

Return Value

This function returns 0 (US_OK) upon success and —1 upon failure. In the event of failure, the external variable Us error contains an appropriate error code. Error codes are defined in userver.h.

Usage Notes

This function is called to set the HTTP latency. The HTTP latency is defined as the maximum allowable age of the OEM data cache for HTTP get requests from the client. If the client browser issue a get request for either the read or the control page, and the data in the OEM data cache is older than the latency period, Micro-server will call the specified callback functions to obtain the current data from the OEM software layer, thus refreshing the cache. The OEM data cache timestamp will be updated accordingly. The default value for the HTTP latency is 5000 ms.

See Also

us_SetTcpLatency()

us_SetTcpLatency()

Micro-Server Initialization

Description

Sets up the Micro-server TCP/IP latency.

Synopsis

#include <userver.h>

int us_SetTcpLatency(long latency)

Arguments

The following arguments are passed to this function:

latency TCP/IP latency in milliseconds

Return Value

This function returns 0 (US_OK) upon success and −1 upon failure. In the event of failure, the external variable Us_error contains an appropriate error code. Error codes are defined in userver.h.

Usage Notes

This function is called to set the TCP/IP latency. The TCP/IP latency is defined as the maximum allowable age of the OEM data cache for TCP/IP requests from the client over the VTG application-to-application protocol. If the client software issues a request for any OEM data variables, and the data in the OEM data cache is older than the latency period, Micro-server will call the specified callback functions to obtain the current data from the OEM software layer, thus refreshing the cache. The OEM data cache timestamp will be updated accordingly. The default value for the TCP/IP latency is 100 ms.

See Also

us_SetHttpLatency()

us_SendTcpResponse()

Micro-Server Communication

Description

Used for TCP/IP dialog directly between remote client and user program, this function sends a response to a message originated by a remote TCP/IP client. This function is only used in the "passthrough" mode.

Synopsis

#include <userver.h>

int us_SendTcpRepsonse(IPADDRESS originator, PACKET *packet)

Arguments

The following arguments are passed to this function:

originator Specifies the IP address of the originating client. This value will have been passed to the user code via the tcpreceive() function called by the Micro-server.

packet Pointer to the packet containing the response to send to the client

Return Value

This function returns 0 (US_OK) upon success and −1 upon failure. In the event of failure, the external variable Us_error contains an appropriate error code. Error codes are defined in userver.h.

See Also

us_SetTcpDialog()

us_SetIpAddress()

Micro-Server Initialization

Description

Micro-server IP address

Synopsis

#include <userver.h>

int us_SetIpAddress(IPADDRESS address)

Arguments

Address is a 32-bit entity containing the IP address of the embedded Micro-server. One byte is allocated to each of the four (4) address components with the most significant 8 bits of address corresponding to the most significant IP address component.

Return Value

This function returns 0 (US_OK) upon success and −1 upon failure. In the event of failure, the external variable Us_error contains an appropriate error code. Error codes are defined in userver.h.

Usage Notes

The value of the IP address passed by the user to Micro-server is typically read from the embedded hardware constituting the execution platform, such as DIP switches. Before initialization, the IP address is set to a default value of 100.100.100.1.

Scc Also

us_SetIpDHCP()

us SetIpDHCP()

Micro-Server Initialization

Description

Initializes the Micro-server IP address via a remote DHCP server

Synopsis

#include <userver.h>

int us_SetIpDHCP(void)

Arguments

None

Rcturn Valuc

This function returns 0 (US_OK) upon success and −1 upon failure. In the event of failure, the external variable Us_error contains an appropriate error code. Error codes are defined in userver.h.

Usage Notes

The value of the IP address is determine by a remote DHCP server. This dynamic address assignment should only be used if the Micro-server-enabled device is accessed via a name resolved by a local DNS server.

See Also

us SetIpAddress()

us_SetMfgData()

Micro-Server Initialization

Description

Used to provide the Micro-server with manufacturer data. Multiple calls to this function are made to provide such data. Majority of the data is destined for the /discover.htm page of the server.

Synopsis

#include <userver.h>

int us SetMfgData(int id, char *value)

Arguments

The following arguments are passed to this function:

id Identifies the data passed in the value argument to the function. Hcadcr filc "uscrvcr.h" contains manifcst constants used for this value. Because these constants contain additional implicit information, they should always be used. Do not use constant values here. For example, US_MFGADR0 should be used instead of 2.

value Specifies the value of the field identified by id. The format of this argument is always a null-terminated string.

Return Value

This function returns 0 (US_OK) upon success and −1 upon failure. In the event of failure, the external variable Us_error contains an appropriate error code. Error codes are defined in userver.h.

Usage Notes

The following table specifies constant values that could be used with the id argument

US_CAPLTSVR	Local Applet Server name
US_CMNTSVR	Maintenance Server name
US_CMYIPADRS	Device IP address
US_CTIMSVR	Local time server IP name
US_ICOUNTRY	Country code
US_MFGADRS0	Manufacturer address 0
US_MFGADRS1	Manufacturer address 1
US_MFGADRS2	Manufacturer address 2
US_MFGADRS3	Manufacturer address 3
US_MFGAPLURL	Manufacturer client applet server URL
US_MFGDATE	Device date of manufacture
US_MFGDOCURL	Manufacturer documentation URL
US_MFGEMAIL	Manufacturer tech support Email address
US_MFGFAX	Manufacturer fax number
US MFGGENER	Manufacturer generic device description
US_MFGLNKMAIL	Manufacturer tech support Email link
US_MFGLOGO	Link to Manufacturer logo
US_MFGMODEL	Manufacturer device model number
US_MFGMSID	Manufacturer Micro-server License ID
US_MFGNAME	Manufacturer Name
Us_MFGPGMVER	Manufacturer Program Version
US_MFGSERNO	Manufacturer device serial number
US_MFGTEL	Manufacturer telephone number
US_MFGURL	Manufacturer general URL
US_MGGBACK	Manufacturer background

us_SetReadData()

Micro-Server Initialization

Description

Micro-server with data to be published to the network. The data is referred to as "read" since it is read by the client. Multiple calls to this function are made to provide such data, with one call made for each of the published data points. Majority of the data is destined for the /read.him or /default.htm page of the server.

Synopsis

#includc <uscrvcr.h>

int us_SetReadData(READDATA *ptr)

Arguments

The following arguments are passed to this function:

ptr Is a pointer to a READDATA definition structure initialized by the caller before the call to us_SetReadData(). Header file "userver.h" contains a declaration for this structure.

Return Value

This function returns 0 (US OK) upon success and −1 upon failure. In the event of failure, the external variable Us_error contains an appropriate error code. Error codes are defined in userver.h.

Usage Notes

The calling program passes a pointer to a data structure containing the description of the read data field to be published. The data structure is declared as:

```
typedef struct    readdata
  {
  char *description;     /* describes the parameter */
  char *units;           /* specifies the measurement units */
  char (*callback)();    /* call back function used to get value */
  char *lowend;          /* the low end of parameter value span */
  char *highend;         /* the low end of parameter value span */
  char *lowlimit;        /* recommended low limit for parameter value */
  char *highlimit;       /* recommended high limit for parameter value */
  char *nominal;         /* recommended nominal value for parameter */
  } READDATA;
```

The callback element of the READDATA structure contains a pointer to a user-provided function that returns the current value of the data. The data is returned as a character string.

Note that the lowlimit, highlimit and nominal strings are advisory in nature. If the user program chooses not to specify them, they may be left blank (by assigning 0-length null terminated strings to corresponding structure elements.

The Micro-server makes private copies of the user data, hence the READDATA structure may be re-used with a subsequent call to us_SetReadData().

See Also

us_SetReadData()

us_SetWriteData()

Micro-Server Initialization

Description

Configures the Micro-server with control data to be published and accepted from the clients. The data is referred to as "write" since it can be written to the OEM layer by the remote client. Multiple calls to this function are made to provide such data, with one call made for each of the published data points. Majority of the data is destined for the /control.htm page of the server.

Synopsis

#include <userver.h>

int us_SetWriteData(WRITEDATA *ptr)

Arguments

The following arguments are passed to this function:

ptr Is a pointer to a WRITEDATA definition structure initialized by the caller before the call to us_SetWriteData(). Header file "userver.h" contains a declaration for this structure.

Return Value

This function returns 0 (US_OK) upon success and −1 upon failure. In the event of failure, the external variable Us_error contains an appropriate error code. Error codes are defined in userver.h.

Usage Notes

The calling program passes a pointer to a data structure containing the description of the read data field to be published. The data structure is declared as:

```
typedef struct    readdata
  {
  char *description;      /* describes the parameter */
  char *units;            /* specifies the measurement units */
  char (*rdcallback)();   /* call back function used to get value */
  char (*wrcallback)();   /* call back function used to set value */
  char *lowend;           /* the low end of parameter value span */
  char *highend;          /* the low end of parameter value span */
  char *lowlimit;         /* recommended low limit for parameter
                          value */
  char *highlimit;        /* recommended high limit for parameter
                          value */
  char *nominal;          /* recommended nominal value for
                          parameter */
  } READDATA;
```

The callback element of the WRITEDATA structure contains a pointer to a user-provided function that returns the current value of the data. The data is always passed as a character string.

Note that the lowlimit, highlimit and nominal strings are advisory in nature. If the user program chooses not to specify them, they may be left blank (by assigning 0-length null terminated strings to corresponding structure elements.

The Micro-server makes private copies of the user data, hence the WRITEDDATA structure may be re-used with a subsequent call to us SetWriteData().

See Also

us_SetReadData()

us_SetTcpDialog()

Micro-Server Initialization

Description

Initializes TCP/IP dialog directly between remote client and user program. This is necessary only in the "passthrough" mode.

Synopsis

#include <userver.h>

int us_SetTcpDialog(int port, void (*tcpreceive)())

Arguments

The following arguments are passed to this function:

portid Specifies the port number to be used in direct communications with the client. Keep in mind that the port may not conflict with the two primary ports are already used: US_SERVERPORT used by the Micro-server for handling HTTP protocol traffic as well US_TCPPORT used by the Micro-server for handling transactions involving client-side generic applets.

tcpreceive Specifies the address to an OEM function of type void, used to write pass a TCP packet received by Micro-server to the OEM software layer. Whenever a TCP packet is received on the specified port, Micro-server calls this function to pass the packet to the OEM software. Micro-server assumes that the function is used as follows:

IPADDRESS originator;

PACKET *packet;

tcpreceive(originator, packet)

Return Value

This function returns 0 (US_OK) upon success and −1 upon failure. In the event of failure, the external variable Us error contains an appropriate error code. Error codes are defined in userver.h.

Usage Notes

The user program is responsible for providing the tcpreceive() function if this feature will be used.

See Also

us_SendTcpResponse()

us_WriteMaintLog()

Micro-Server Operation

Description

Micro-server has been remotely configured with a name or IP address of a local maintenance server, the entry is made in the server log via an appropriate TCP/IP transaction.

See also

us_deleteMaintLog()

We claim:

1. A system for providing information about a remote device to a client workstation, the system comprising:

a micro-server for transmitting the information to the client workstation, the micro-server defining an application programming interface for interfacing with the remote device to access the information from the remote device and for abstracting the details of transmitting the information to the client workstation, the remote device initializing the micro-server by providing, via a function call defined by the application programming interface, at least one pointer to at least one callback function for accessing the information, the micro-server accessing the information by calling the at least one callback function.

2. The system of claim 1 wherein the micro-server comprises a TCP/IP application programming interface for accessing a TCP/IP protocol stack.

3. The system of claim 1 further comprising: a system services application programming interface for providing the micro-server with access to system services from the remote device.

4. The system of claim 1 wherein the micro-server comprises an HTTP protocol server for satisfying interactive HTTP requests from the remote client workstation.

5. The system of claim 4 wherein the micro-server comprises a synthesized HTML server for providing a dynamic copy of an HTML version of a web page served by the HTTP protocol server.

6. The system of claim 1 further comprising: a data image for caching data from the remote device for reducing the number of application programming interface function calls made by the micro-server to access remote device information.

7. The system of claim 1 wherein the micro-server comprises: a TCP/IP based server for satisfying TCP/IP based requests from the client workstation.

8. The system of claim 1 wherein the micro-server comprises: a hyperlink to a website associated with the remote device.

9. The system of claim 1 wherein the micro-server comprises: a web site associated with the remote device.

10. The system of claim 9 wherein the web site comprises: a home page for publishing the remote device's parametric data.

11. The system of claim 9 wherein the web site comprises: a control page for providing authorized individuals access to the remote device's control functions.

12. The system of claim 9 wherein the web site comprises: a maintenance page for providing access to the remote device's maintenance data.

13. The system of claim 12 wherein the maintenance page comprises: a hyperlink to online maintenance documentation.

14. The system of claim 9 wherein the web site comprises: a maintenance page for providing access to the remote device's maintenance functions.

15. The system of claim 9 wherein the web site comprises: an administration page for allowing authorized individuals to set operating characteristics of the remote device.

16. The system of claim 9 wherein the web site comprises: a discover page for supplying information to the client workstation during automatic discovery.

17. The system of claim 1 further comprising: a default applet server for providing default applets to the client workstation for interfacing with the remote device from the client workstation.

18. The system of claim 1 further comprising: a time server for providing current time information to the system, the time information being maintained based at least in part upon universal coordinated time.

19. The system of claim 1 further comprising: a maintenance server for providing non-volatile storage for the remote device's maintenance records, the non-volatile storage being hyperlinked from the remote device.

20. The system of claim 1 further comprising: an auto-discovery and view server for automatically

detecting a micro-server-enabled device being coupled to an interface and

publishing micro-server-enabled device data to the client workstation.

21. The system of claim 1 further comprising: a local applet server for retrieving device-specific applets from a server associated with the remote device and for making the applets available to the client workstation.

22. The system of claim 1 further comprising: a browser for interacting with the remote device.

23. The system of claim 1 further comprising: a client-side application programming interface for providing the client workstation with a software interface to the remote device.

24. The system of claim 1 wherein at least a portion of the information is organized as addressable variables.

25. A remote device capable of providing information about itself to a client workstation, the remote device comprising:

a micro-server for transmitting the information to the client workstation while abstracting communication protocol details from the remote device's control software;

an application programming interface for providing an interface between the remote device's control software and the micro-server, the control software initializing the micro-server by providing, via a function call defined by the application programming interface, at least one pointer to at least one callback function for accessing the information, the micro-server accessing the information by calling the at least one callback function;

a hardware Ethernet interface; and

a TCP/IP protocol stack for interfacing between the micro-server and the hardware Ethernet interface.

26. The remote device of claim 25 further comprising: a computer capable of supporting a standard operating system environment.

27. A micro-circuit board for providing information about a remote device to a client workstation, the micro-circuit board comprising:

a micro-server for transmitting the information to the client workstation while abstracting communication protocol programming details from the remote device's control software;

an application programming interface for providing an interface between the remote device's control software and the micro-server, the control software initializing the micro-server by providing, via a function call defined by the application programming interface, at least one pointer to at least one callback function for accessing the information the micro-server accessing the information by calling the at least one callback function;

a hardware Ethernet interface; and

a TCP/IP protocol stack for interfacing between the micro-server and the hardware Ethernet interface.

28. A system for providing information about a remote device to a client workstation, the system comprising:

a first processor for running the remote device's control software and a first side of an application programming interface ("API");

a second processor for running micro-server software and abstracting communication protocol details from the first processor by running a second side of the API, the control software initializing the micro-server by providing, via a function call defined by the API, at least one pointer to at least one callback function for accessing the information, the micro-server accessing the information by calling the at least one callback function and transmitting the information to the client workstation, the second processor accessing a TCP/IP stack to interface with the hardware Ethernet interface; and

a hardware interface between the first processor and the second processor.

29. A system for providing information about a remote device to a client workstation, the system comprising:

a processor for running remote device control software and a first side of an application programming interface ("API"), the processor being mounted on a PC circuit board, the PC circuit board being insertable into a computer;

a computer for abstracting communication protocol details from the processor by running micro-server software including a second side of the API, the control software initializing the micro-server by providing, via a function call defined by an application programming interface, at least one pointer to at least one callback function for accessing the information, the micro-server accessing the information by calling the at least one callback function and transmitting the information to the client workstation, the computer having a hardware Ethernet interface and a TCP/IP stack for interfacing with the hardware Ethernet interface; and

a hardware interface between the processor and the computer.

30. A method for providing information about a remote device to a client workstation comprising the steps of:

providing to a micro-server, via a function call defined by an application programming interface, at least one pointer to at least one callback function;

accessing the information from the first device via the at least one callback function;

organizing the information into a format compatible with a communication protocol in preparation for making the information available to the client workstation;

making the information available to the client workstation; and

abstracting the communication protocol from the remote device.

31. A system for accessing remote device data and communicating the data to a client workstation, the system comprising:

a remote device having an original equipment manufacturer ("OEM") software component for controlling operation of the remote device and a micro-server software component;

the micro-server software component for transmitting the remote device data to the client workstation, the micro-server software component having an original equipment manufacturer application programming interface ("OEM API") for allowing the OEM software to initialize the micro-server software, the initialization including the OEM software providing one or more callback functions to the micro-server software component to allow the micro-server software component to access OEM software component data through one or more functions defined by the OEM API, the OEM API abstracting micro-server software component implementation of networking protocol details from the OEM software component.

32. The system of claim **31** wherein the micro-server software component comprises non-blocking threads for returning processor control to the OEM software component without delays associated waiting for external events to occur.

33. The system of claim **31** wherein the micro-server software component is a binary software library linked to the OEM application.

34. The system of claim **31** wherein the micro-server software component includes a client-side refresh mode of operation that automatically updates a page displayed at the client workstation at fixed time intervals.

35. The system of claim **31** wherein the micro-server software component comprises an OEM data cache component for caching OEM data accessed by the micro-server software component thereby reducing overhead associated with frequent OEM API function calls.

* * * * *

US006581088B1

(12) **United States Patent**
Jacobs et al.

(10) **Patent No.:** **US 6,581,088 B1**
(45) **Date of Patent:** **Jun. 17, 2003**

(54) SMART STUB OR ENTERPRISE JAVA™ BEAN IN A DISTRIBUTED PROCESSING SYSTEM

(75) Inventors: **Dean B. Jacobs**, Berkeley, CA (US); **Eric M. Halpern**, San Francisco, CA (US)

(73) Assignee: **Beas Systems, Inc.**, San Jose, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/405,260**

(22) Filed: **Sep. 23, 1999**

Related U.S. Application Data

(60) Provisional application No. 60/107,167, filed on Nov. 5, 1998.

(51) **Int. Cl.**^{7} **G06F 9/00**

(52) **U.S. Cl.** **709/105**; 709/100; 709/203

(58) **Field of Search** 709/100, 102, 709/103, 104, 107, 108, 310, 315, 316, 317, 328, 318, 319

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,465,365	A		11/1995	Winterbottom	707/104
5,475,819	A		12/1995	Miller et al.	395/200.03
5,564,070	A		10/1996	Want et al.	455/53.1
5,623,666	A		4/1997	Pike et al.	707/200
5,692,180	A		11/1997	Lee	707/10
5,701,484	A		12/1997	Artsy	709/242
5,701,502	A		12/1997	Baker et al.	709/201
5,764,982	A		6/1998	Madduri	707/10
5,774,660	A	*	6/1998	Brendel et al.	395/200.31
5,790,548	A		8/1998	Sistanizadeh et al.	707/10
5,794,006	A		8/1998	Sanderman	707/10
5,805,804	A		9/1998	Laursen et al.	395/200.02
5,819,019	A		10/1998	Nelson	707/10
5,819,044	A		10/1998	Kawabe et al.	395/200.56
5,842,219	A		11/1998	High, Jr. et al.	707/103
5,850,449	A		12/1998	McManis	703/161
5,862,331	A		1/1999	Herriot	395/200.49
5,901,227	A		5/1999	Perlman	380/21
5,961,582	A		10/1999	Gaines	709/1
5,966,702	A		10/1999	Fresko et al.	707/1
5,974,441	A		10/1999	Rogers et al.	709/200
5,983,233	A		11/1999	Potonniee	707/103
5,983,351	A		11/1999	Glogau	713/201
5,999,988	A		12/1999	Pelegri-Llopart et al.	709/330
6,003,065	A		12/1999	Yan et al.	709/201
6,006,264	A	*	12/1999	Colby et al.	709/226
6,016,505	A		1/2000	Badovinatz et al.	709/205
6,144,944	A	*	11/2000	Kurtzman, II et al.	705/14

OTHER PUBLICATIONS

Brewing Distributed Applications With Java ORB Tools, Dec. 1996.*

Sun Microsystems, JavaBeans, Graham Hamilton, Jul. 1997.*

* cited by examiner

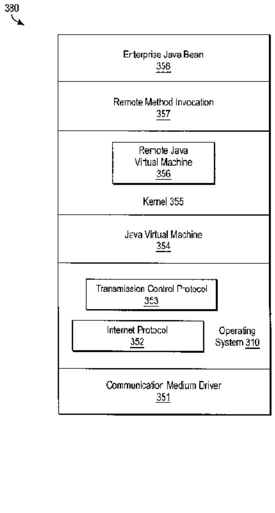
Primary Examiner—Majid Banankhah

(74) *Attorney, Agent, or Firm*—Fliesler, Dubb, Meyer & Lovejoy LLP

(57) **ABSTRACT**

A clustered enterprise Java™ distributed processing system is provided. The distributed processing system includes a first and a second computer coupled to a communication medium. The first computer includes a Java™ virtual machine (JVM) and kernel software layer for transferring messages, including a remote Java™ virtual machine (RJVM). The second computer includes a JVM and a kernel software layer having a RJVM. Messages are passed from a RJVM to the JVM in one computer to the JVM and RJVM in the second computer. Messages may be forwarded through an intermediate server or rerouted after a network reconfiguration. Each computer includes a Smart stub having a replica handler, including a load balancing software component and a failover software component. Each computer includes a duplicated service naming tree for storing a pool of Smart stubs at a node. The computers may be programmed in a stateless, stateless factory, or a stateful programming model. The clustered enterprise Java™ distributed processing system allows for enhanced scalability and fault tolerance.

74 Claims, 16 Drawing Sheets



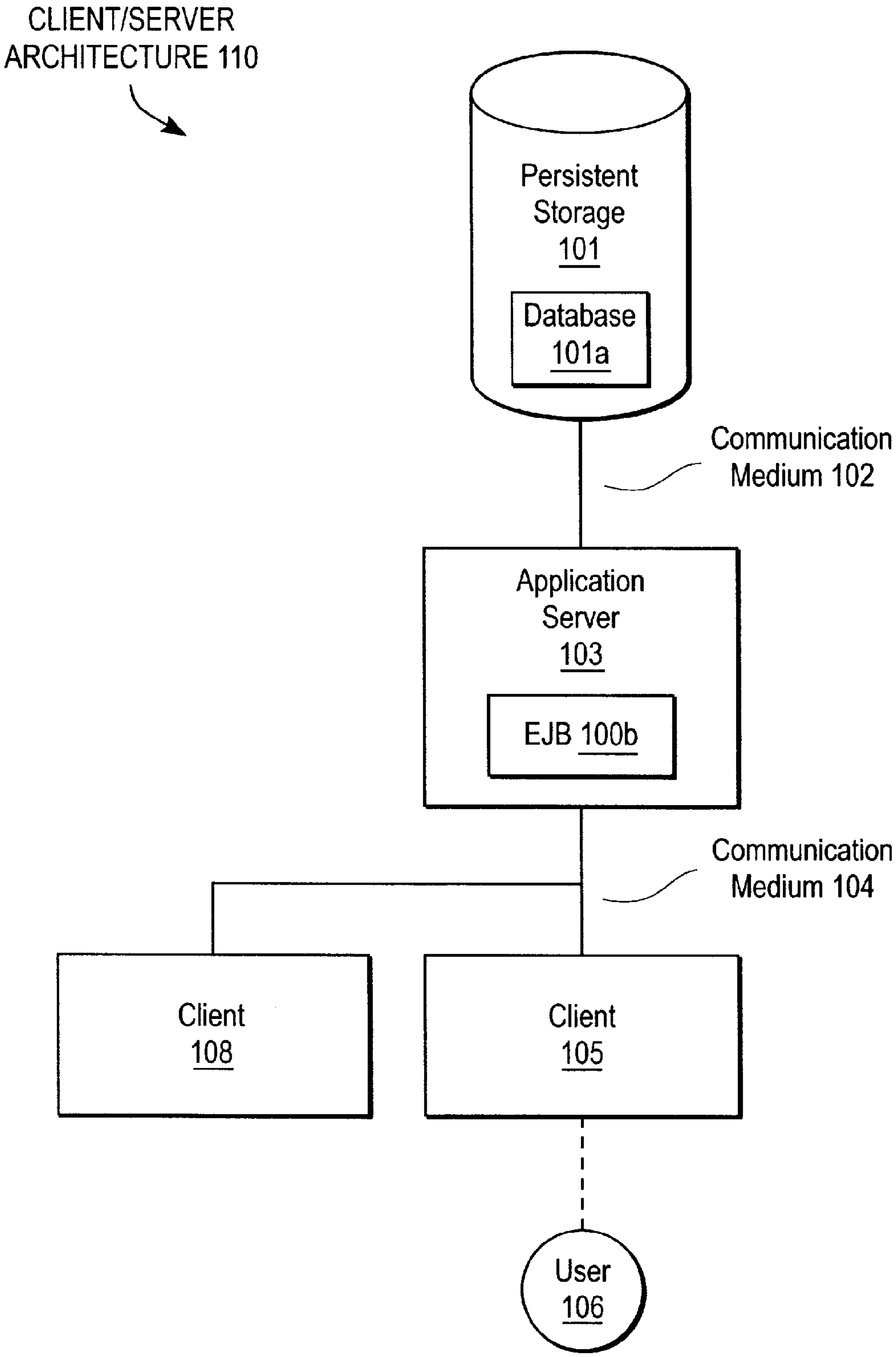


FIG. 1a (PRIOR ART)

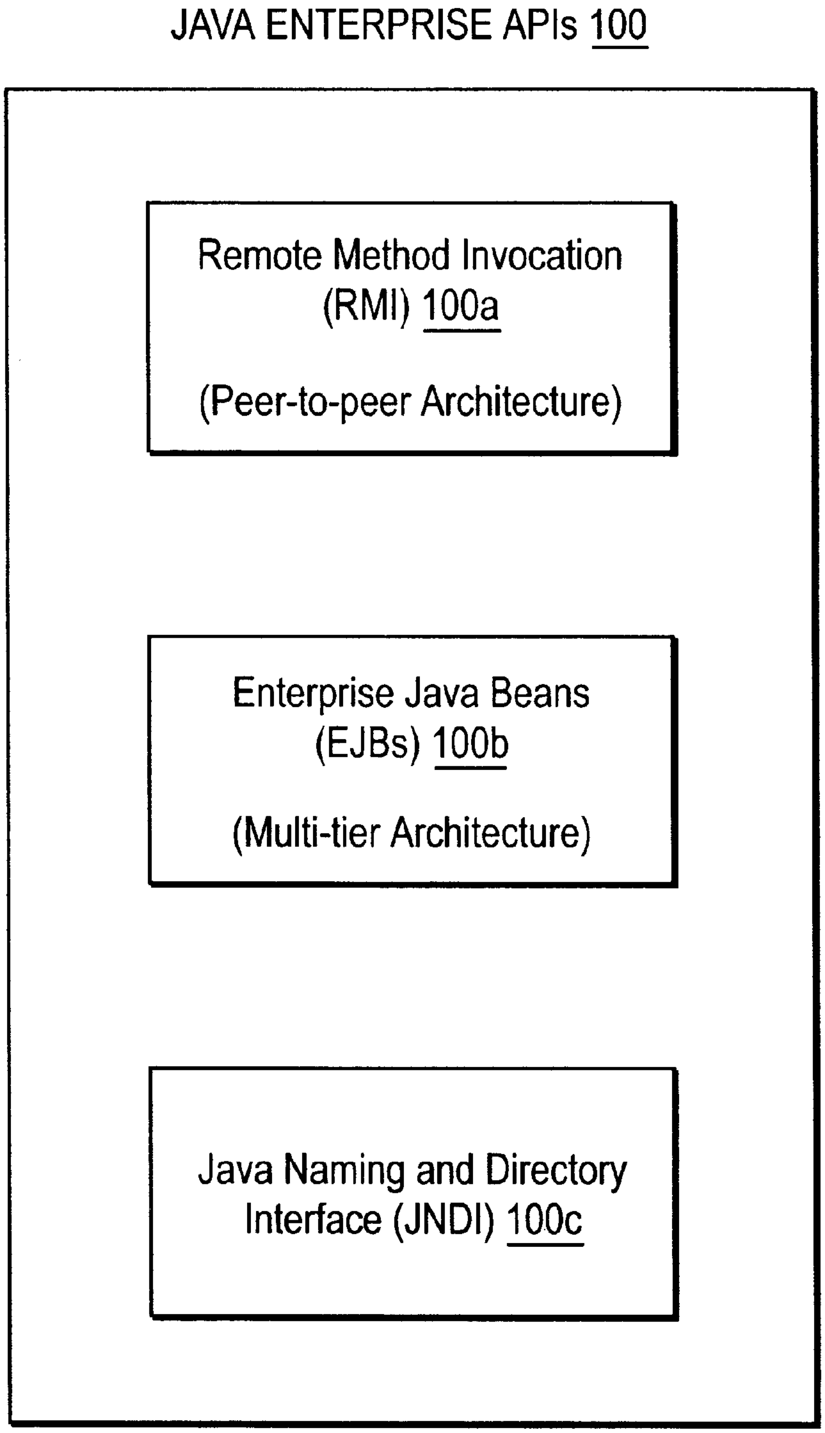

JAVA ENTERPRISE APIs 100
Remote Method Invocation (RMI) 100a
(Peer-to-peer Architecture)
Enterprise Java Beans (EJBs) 100b
(Multi-tier Architecture)
Java Naming and Directory Interface (JNDI) 100c

FIG. 1b (PRIOR ART)

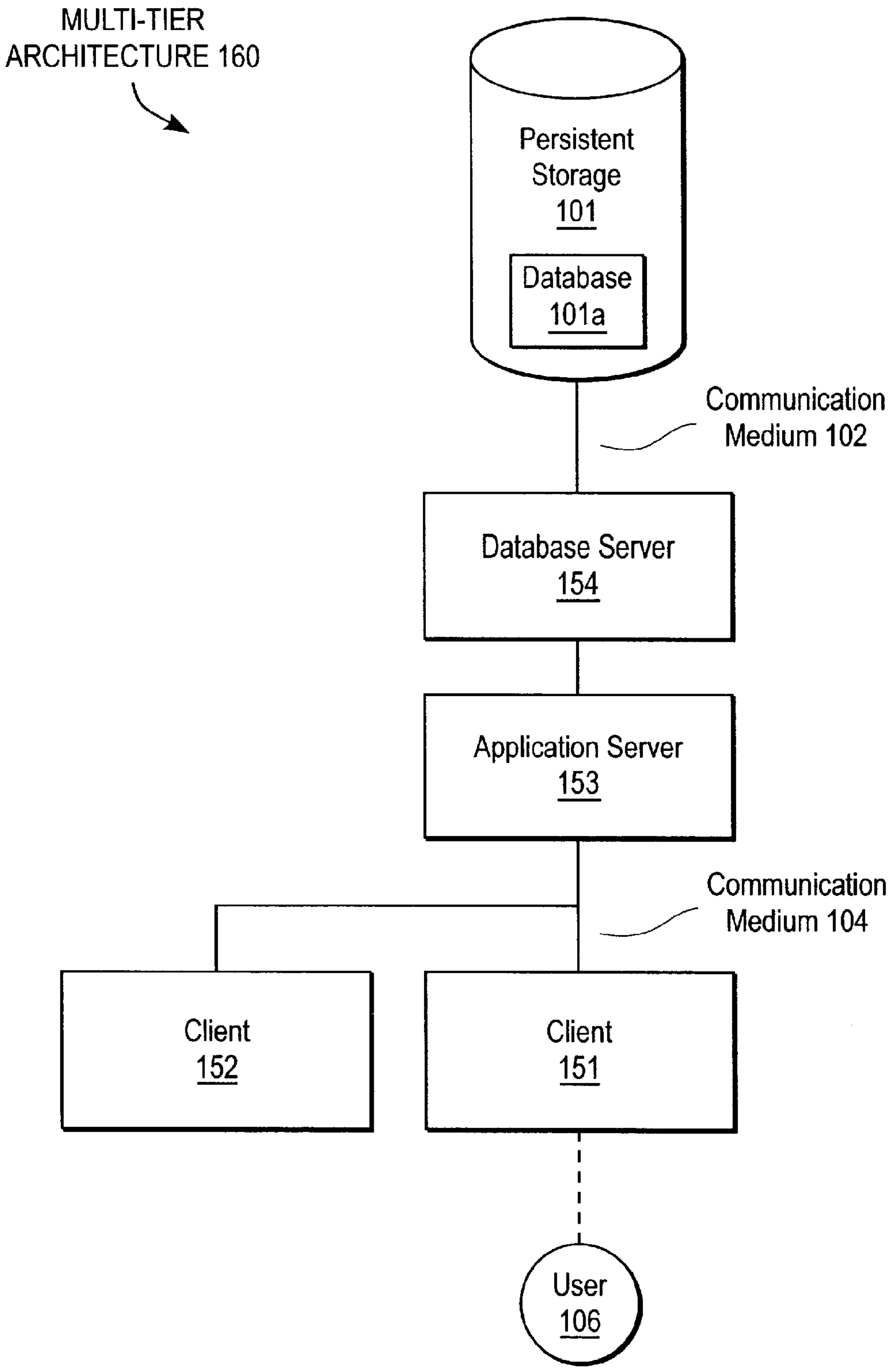


FIG. 1c (PRIOR ART)

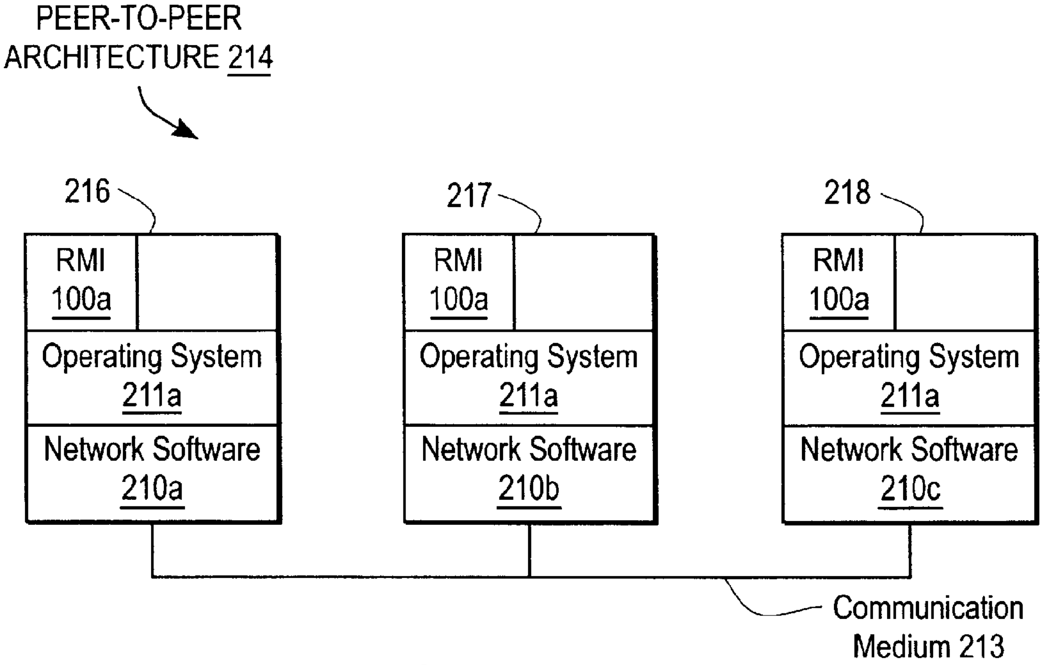


FIG. 2a (PRIOR ART)

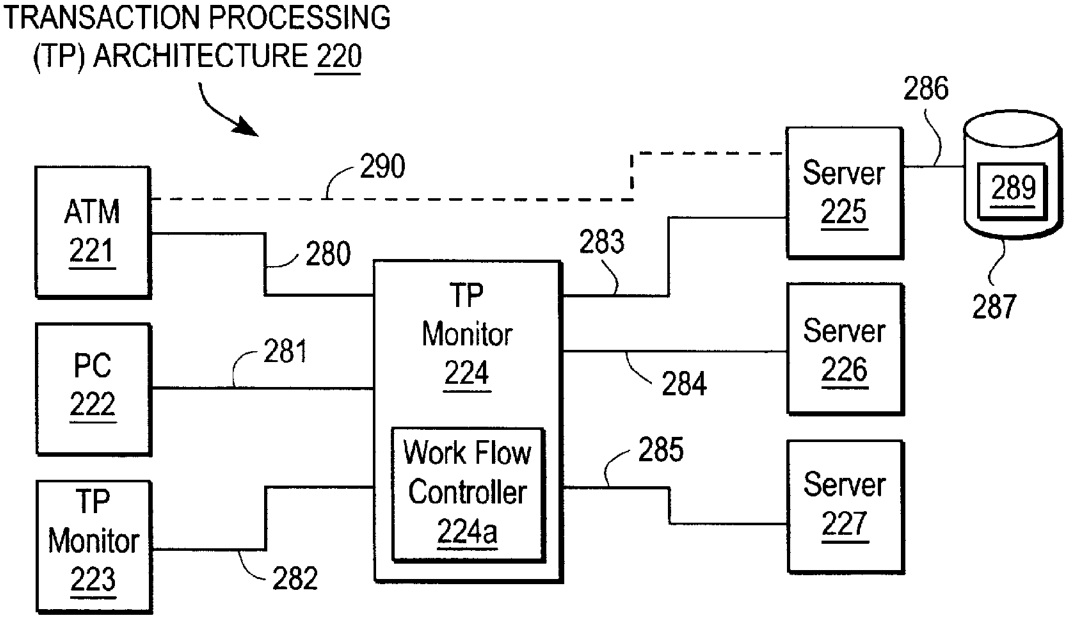


FIG. 2b (PRIOR ART)

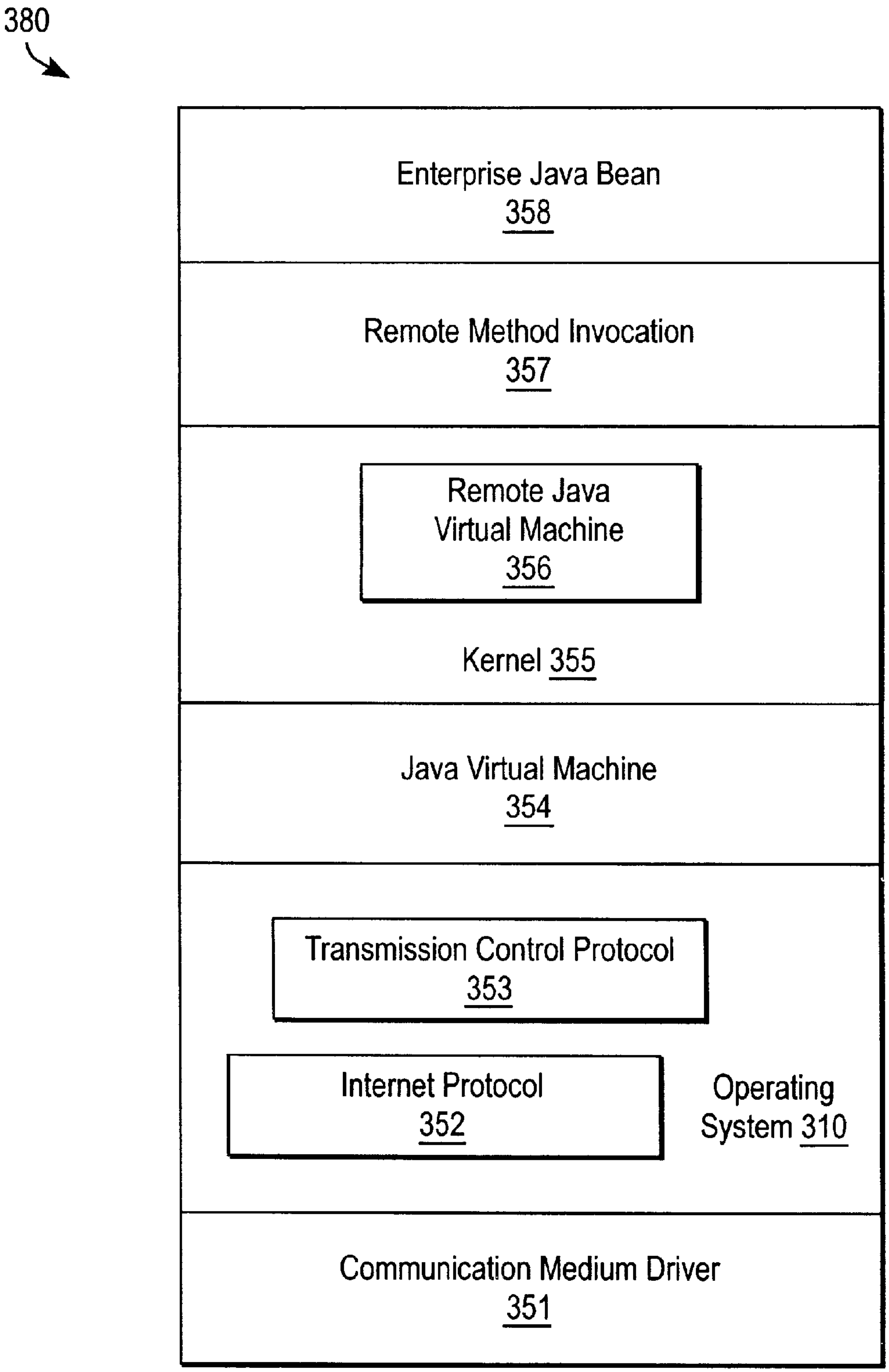

380
Enterprise Java Bean 358
Remote Method Invocation 357
Remote Java Virtual Machine 356
Kernel 355
Java Virtual Machine 354
Transmission Control Protocol 353
Internet Protocol 352
Operating System 310
Communication Medium Driver 351

FIG. 3a

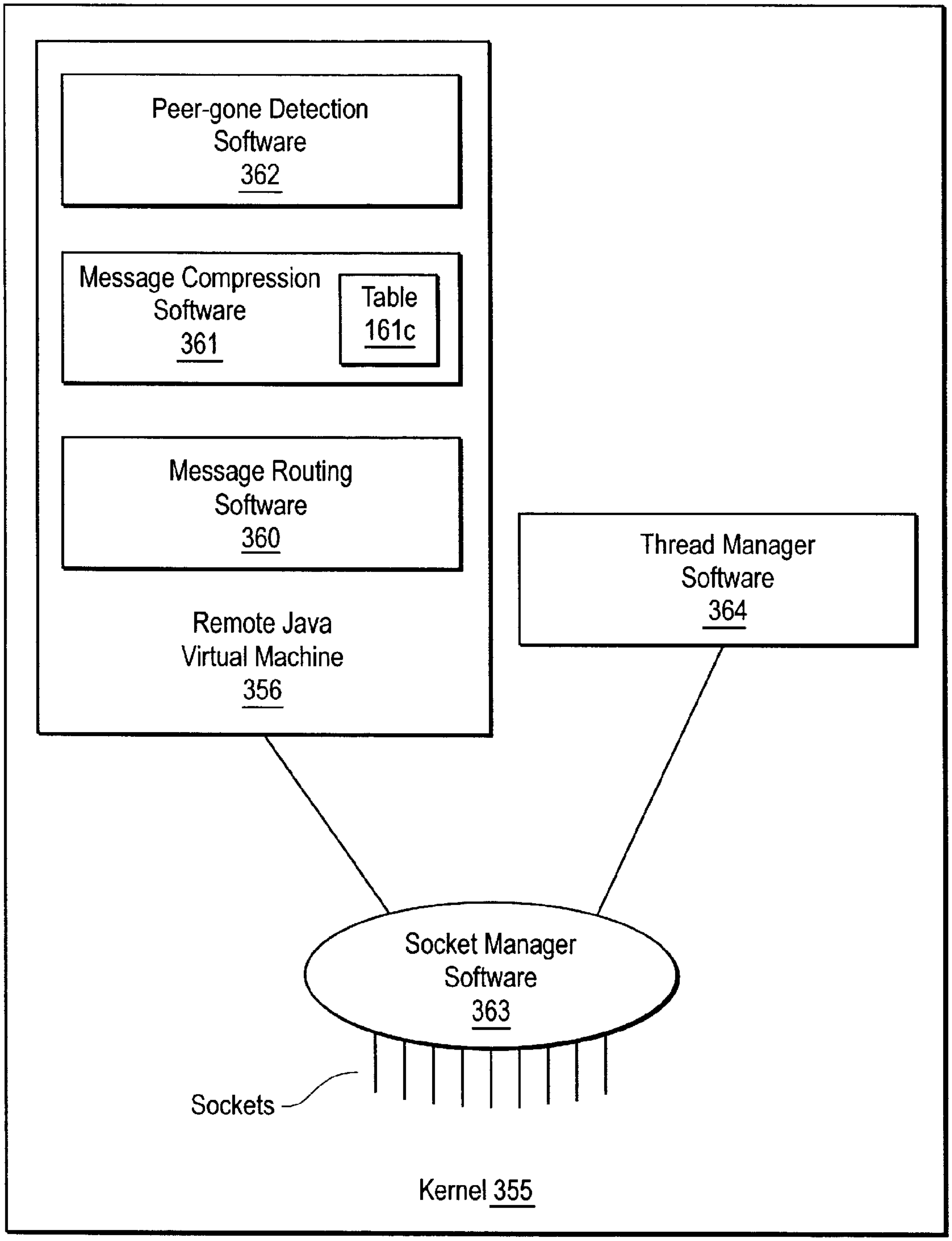

Peer-gone Detection Software 362
Message Compression Software 361
Table 161c
Message Routing Software 360
Remote Java Virtual Machine 356
Thread Manager Software 364
Socket Manager Software 363
Sockets
Kernel 355

FIG. 3b

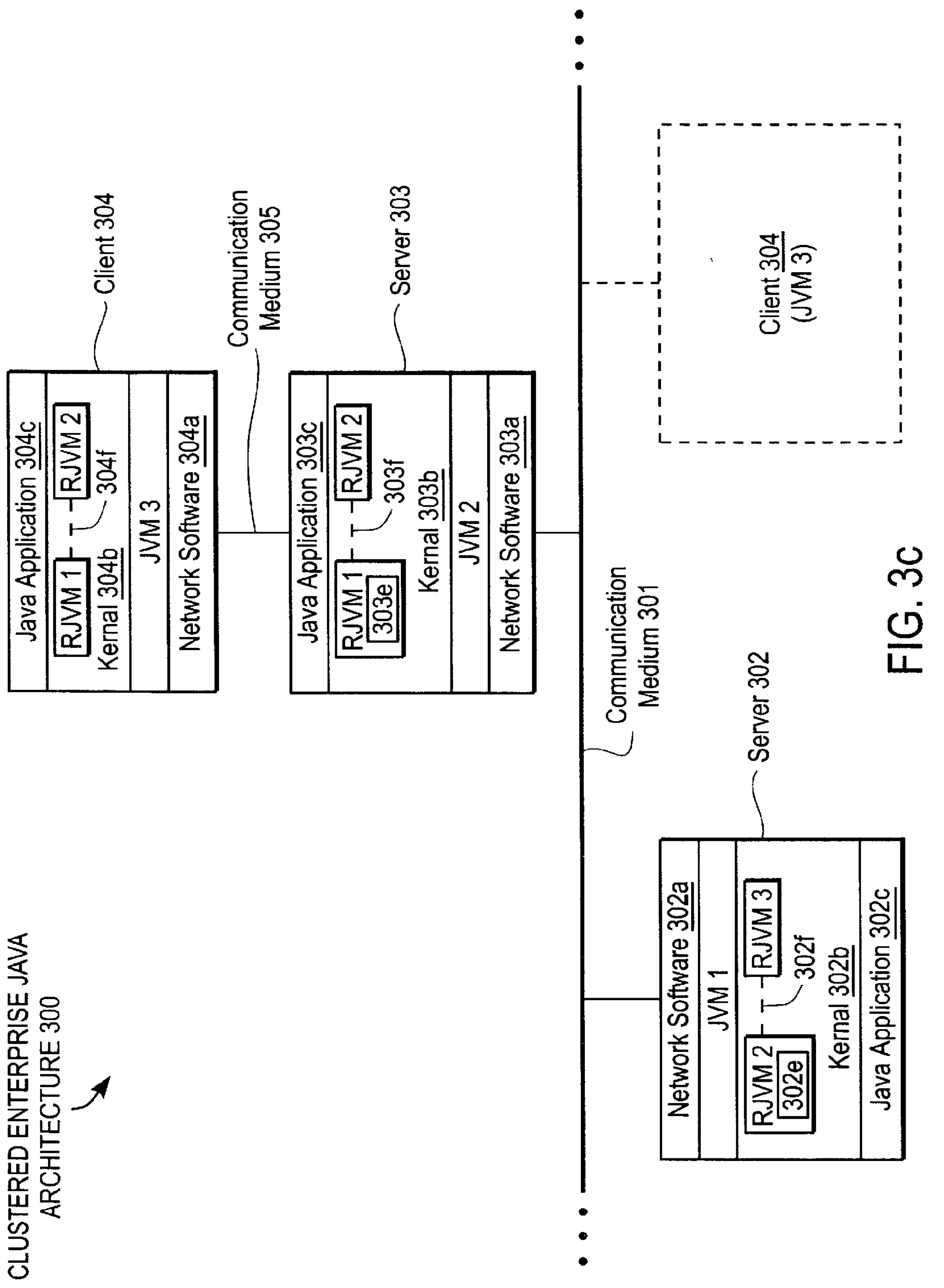

CLUSTERED ENTERPRISE JAVA ARCHITECTURE 300
Client 304
Java Application 304c
RJVM 1
RJVM 2
304f
Kernal 304b
JVM 3
Network Software 304a
Communication Medium 305
Server 303
Java Application 303c
RJVM 1
303e
RJVM 2
303f
Kernal 303b
JVM 2
Network Software 303a
Communication Medium 301
Client 304
(JVM 3)
Server 302
Network Software 302a
JVM 1
RJVM 2
302e
RJVM 3
302f
Kernal 302b
Java Application 302c

FIG. 3c

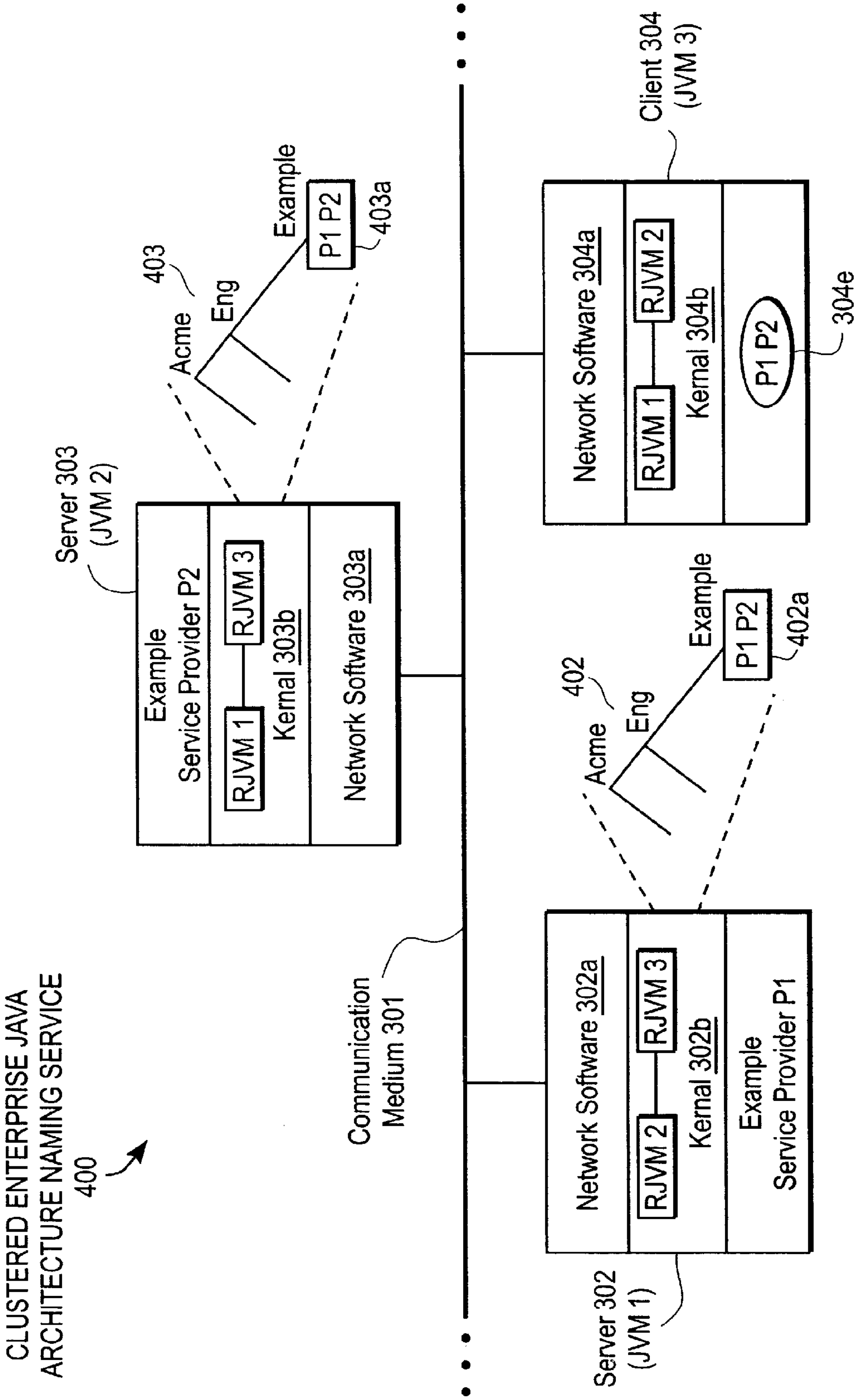

CLUSTERED ENTERPRISE JAVA
ARCHITECTURE NAMING SERVICE
400
Communication
Medium 301
Server 303
(JVM 2)
Example
Service Provider P2
RJVM 1
RJVM 3
Kernal 303b
Network Software 303a
403
Acme
Eng
Example
P1 P2
403a
Client 304
(JVM 3)
Network Software 304a
RJVM 1
RJVM 2
Kernal 304b
P1 P2
304e
Server 302
(JVM 1)
Network Software 302a
RJVM 2
RJVM 3
Kernal 302b
Example
Service Provider P1
402
Acme
Eng
Example
P1 P2
402a

FIG. 4

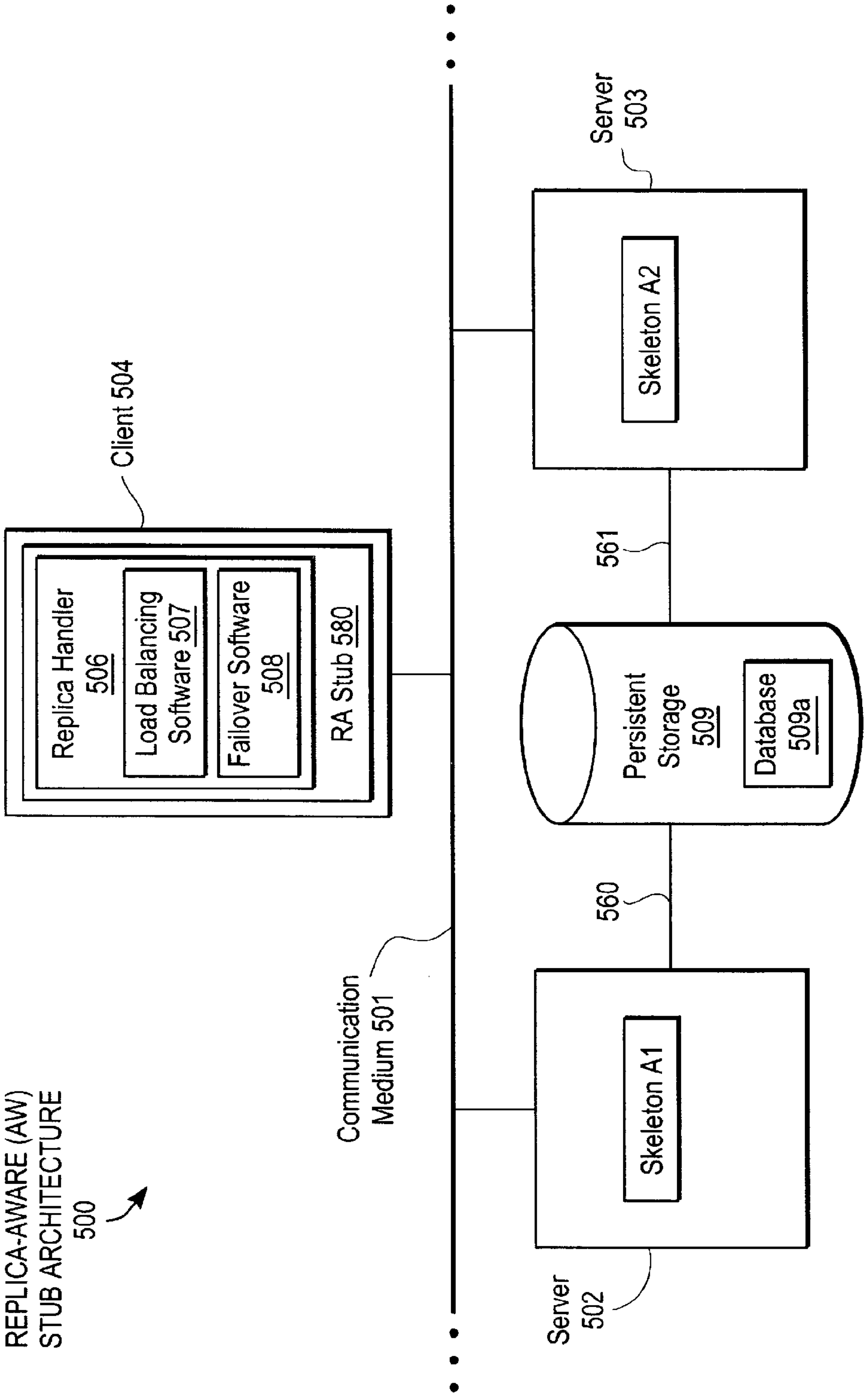

REPLICA-AWARE (AW) STUB ARCHITECTURE
500
Client 504
Replica Handler 506
Load Balancing Software 507
Failover Software 508
RA Stub 580
Communication Medium 501
Server 502
Skeleton A1
560
Persistent Storage 509
Database 509a
561
Skeleton A2
Server 503

FIG. 5a

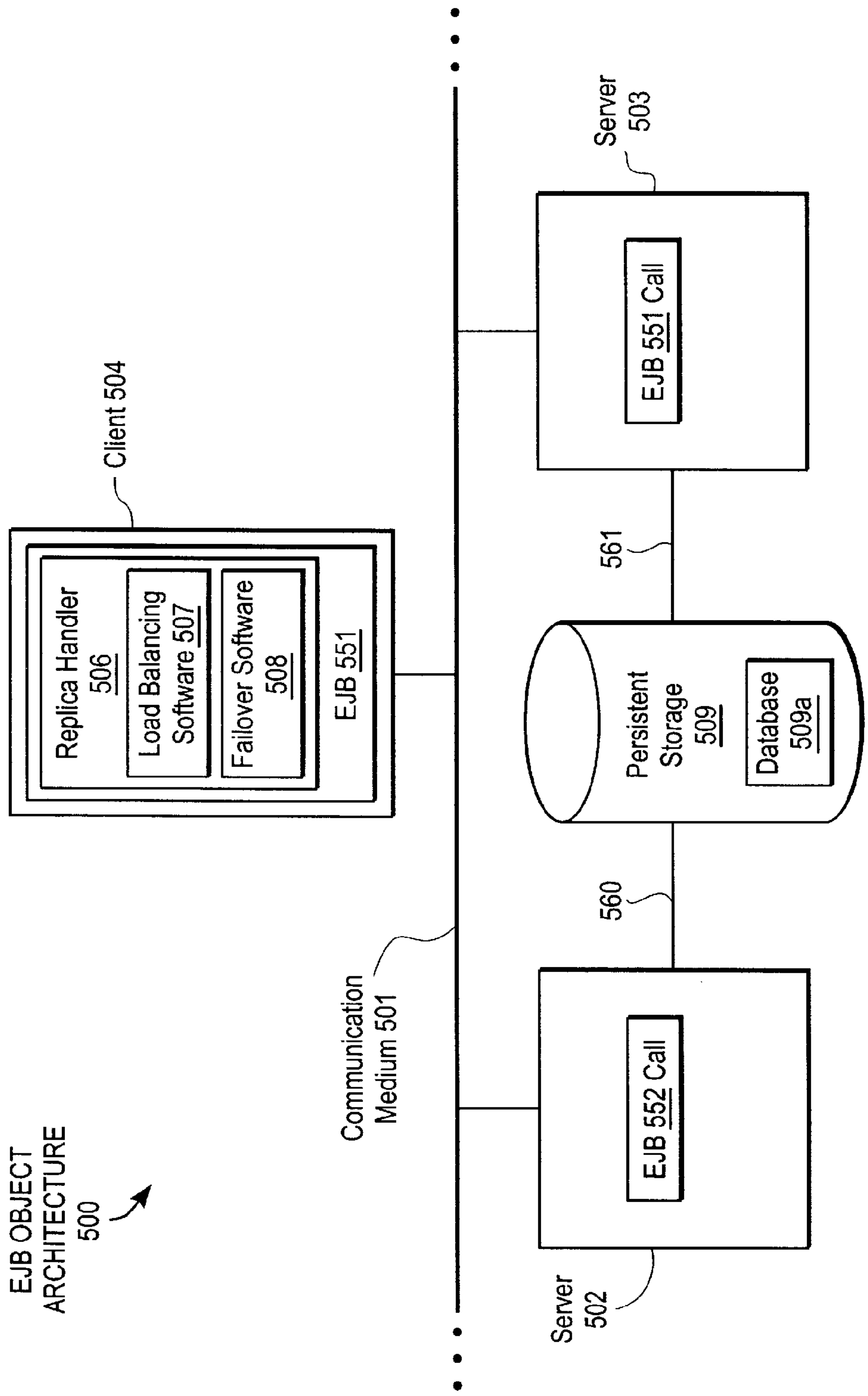

EJB OBJECT ARCHITECTURE 500
Client 504
Replica Handler 506
Load Balancing Software 507
Failover Software 508
EJB 551
Communication Medium 501
Server 502
EJB 552 Call
560
Persistent Storage 509
Database 509a
561
Server 503
EJB 551 Call

FIG. 5b

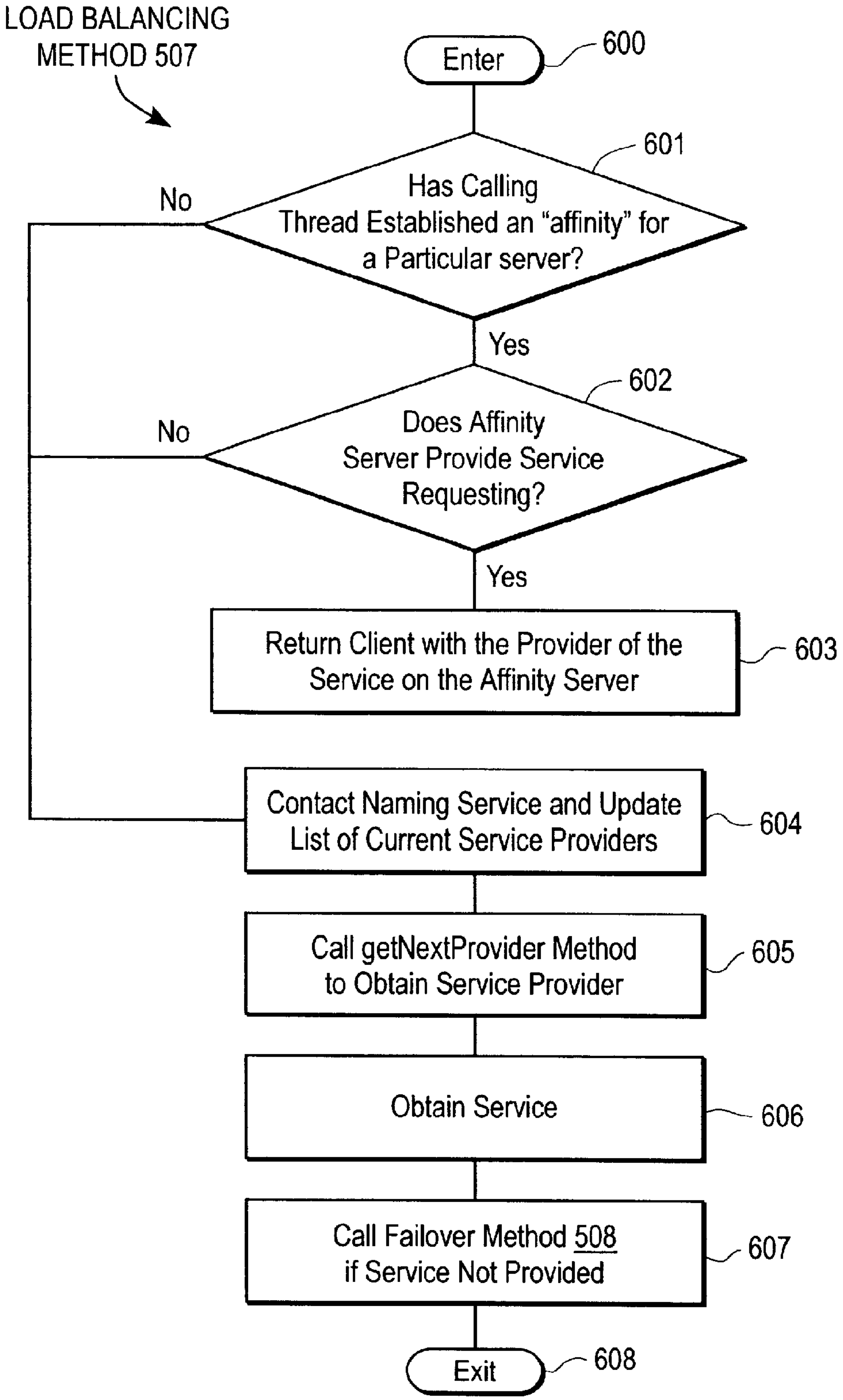

LOAD BALANCING
METHOD 507
Enter
600
601
Has Calling Thread Established an "affinity" for a Particular server?
No
Yes
602
Does Affinity Server Provide Service Requesting?
No
Yes
Return Client with the Provider of the Service on the Affinity Server
603
Contact Naming Service and Update List of Current Service Providers
604
Call getNextProvider Method to Obtain Service Provider
605
Obtain Service
606
Call Failover Method 508 if Service Not Provided
607
Exit
608

FIG. 6a

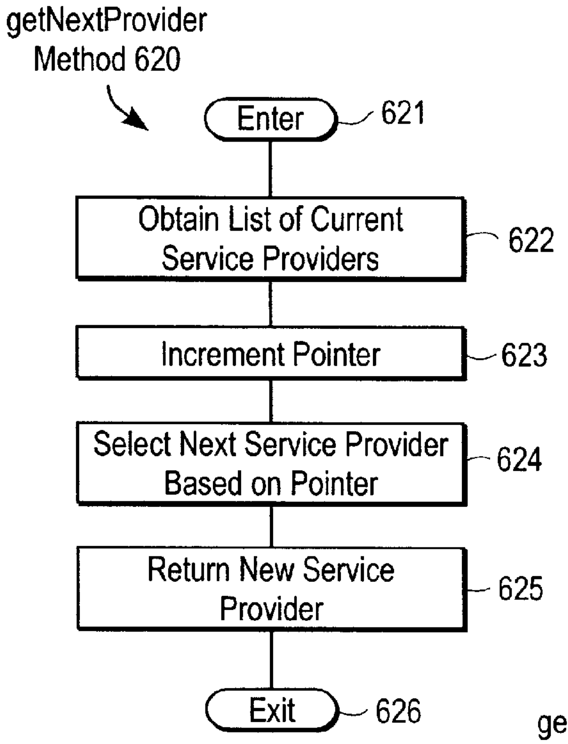

getNextProvider
Method 620
Enter
621
Obtain List of Current Service Providers
622
Increment Pointer
623
Select Next Service Provider Based on Pointer
624
Return New Service Provider
625
Exit
626

FIG. 6b

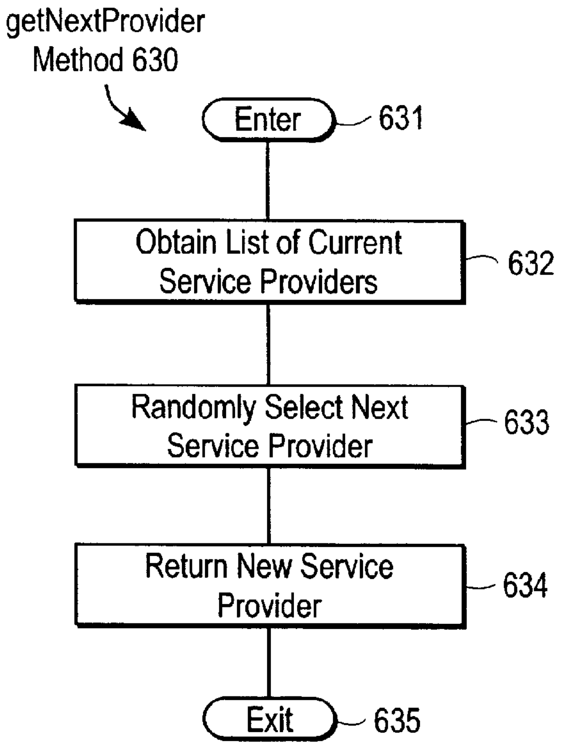

getNextProvider
Method 630
Enter
631
Obtain List of Current Service Providers
632
Randomly Select Next Service Provider
633
Return New Service Provider
634
Exit
635

FIG. 6c

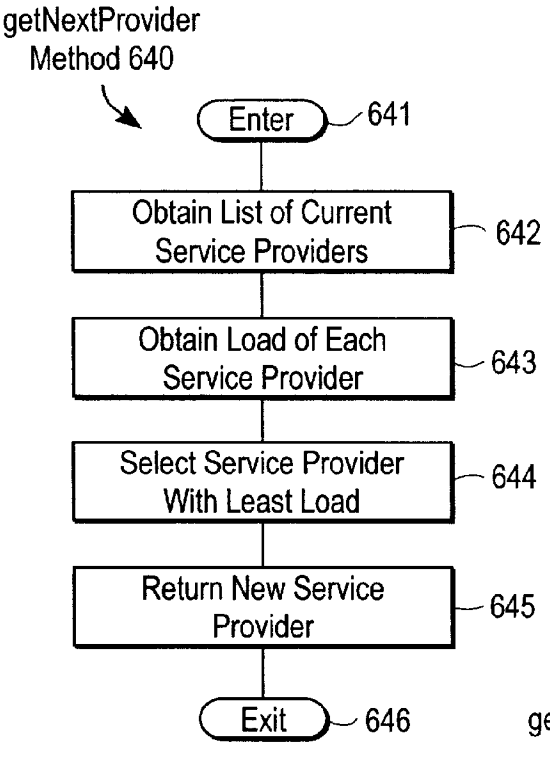

getNextProvider
Method 640
Enter
641
Obtain List of Current Service Providers
642
Obtain Load of Each Service Provider
643
Select Service Provider With Least Load
644
Return New Service Provider
645
Exit
646

FIG. 6d

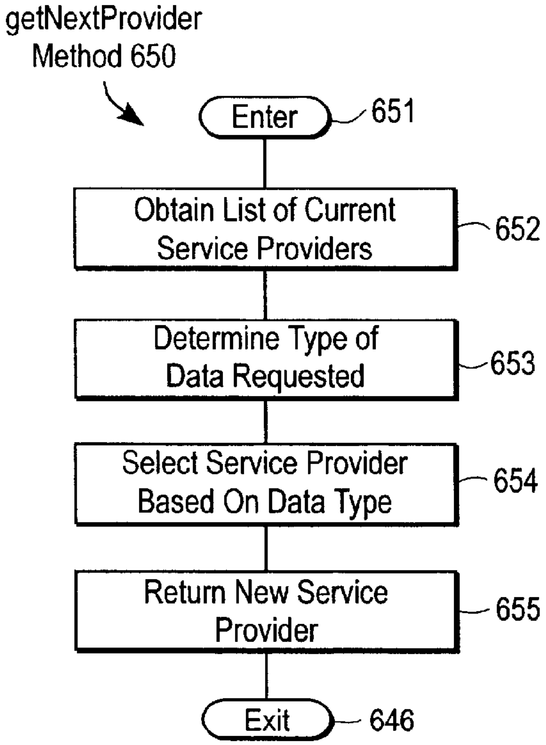

getNextProvider
Method 650
Enter
651
Obtain List of Current Service Providers
652
Determine Type of Data Requested
653
Select Service Provider Based On Data Type
654
Return New Service Provider
655
Exit
646

FIG. 6e

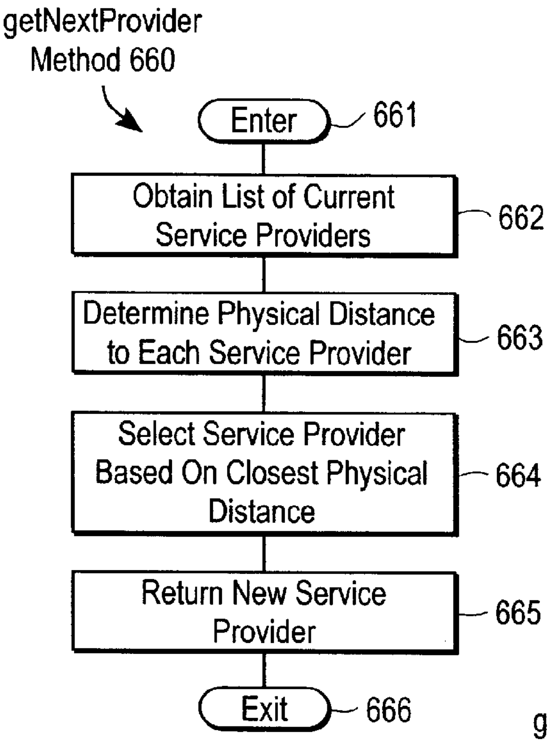

getNextProvider
Method 660
Enter
661
Obtain List of Current Service Providers
662
Determine Physical Distance to Each Service Provider
663
Select Service Provider Based On Closest Physical Distance
664
Return New Service Provider
665
Exit
666

FIG. 6f

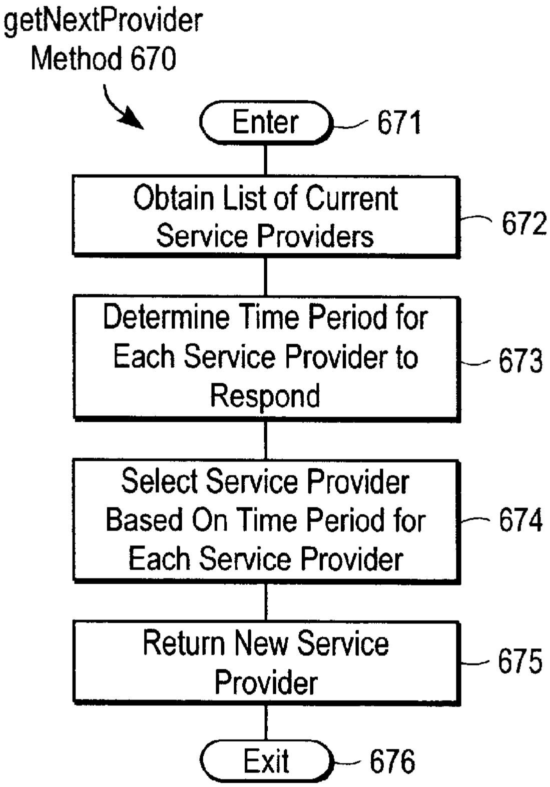

getNextProvider
Method 670
Enter
671
Obtain List of Current Service Providers
672
Determine Time Period for Each Service Provider to Respond
673
Select Service Provider Based On Time Period for Each Service Provider
674
Return New Service Provider
675
Exit
676

FIG. 6g

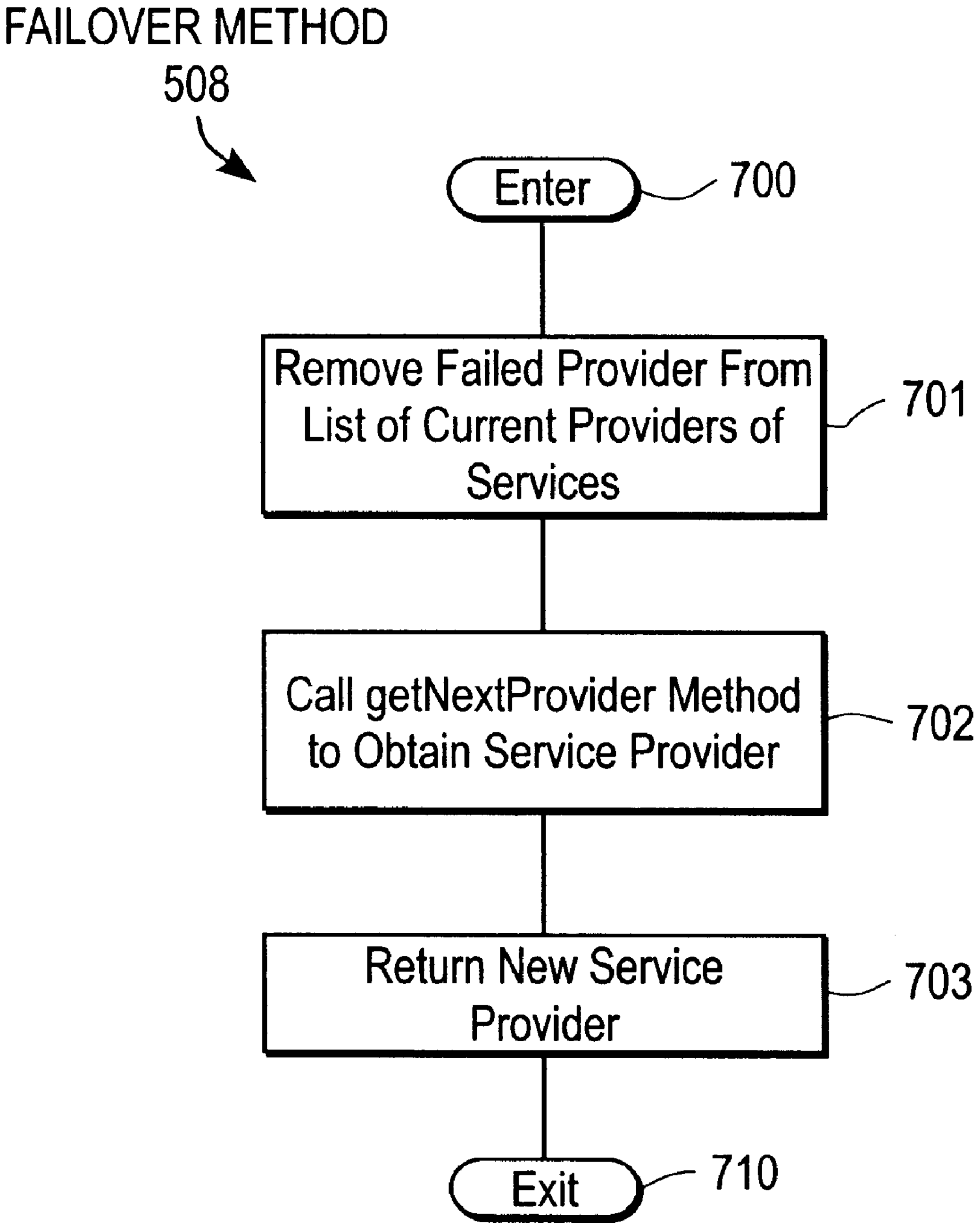

FAILOVER METHOD
508
Enter
700
Remove Failed Provider From List of Current Providers of Services
701
Call getNextProvider Method to Obtain Service Provider
702
Return New Service Provider
703
Exit
710

FIG. 7

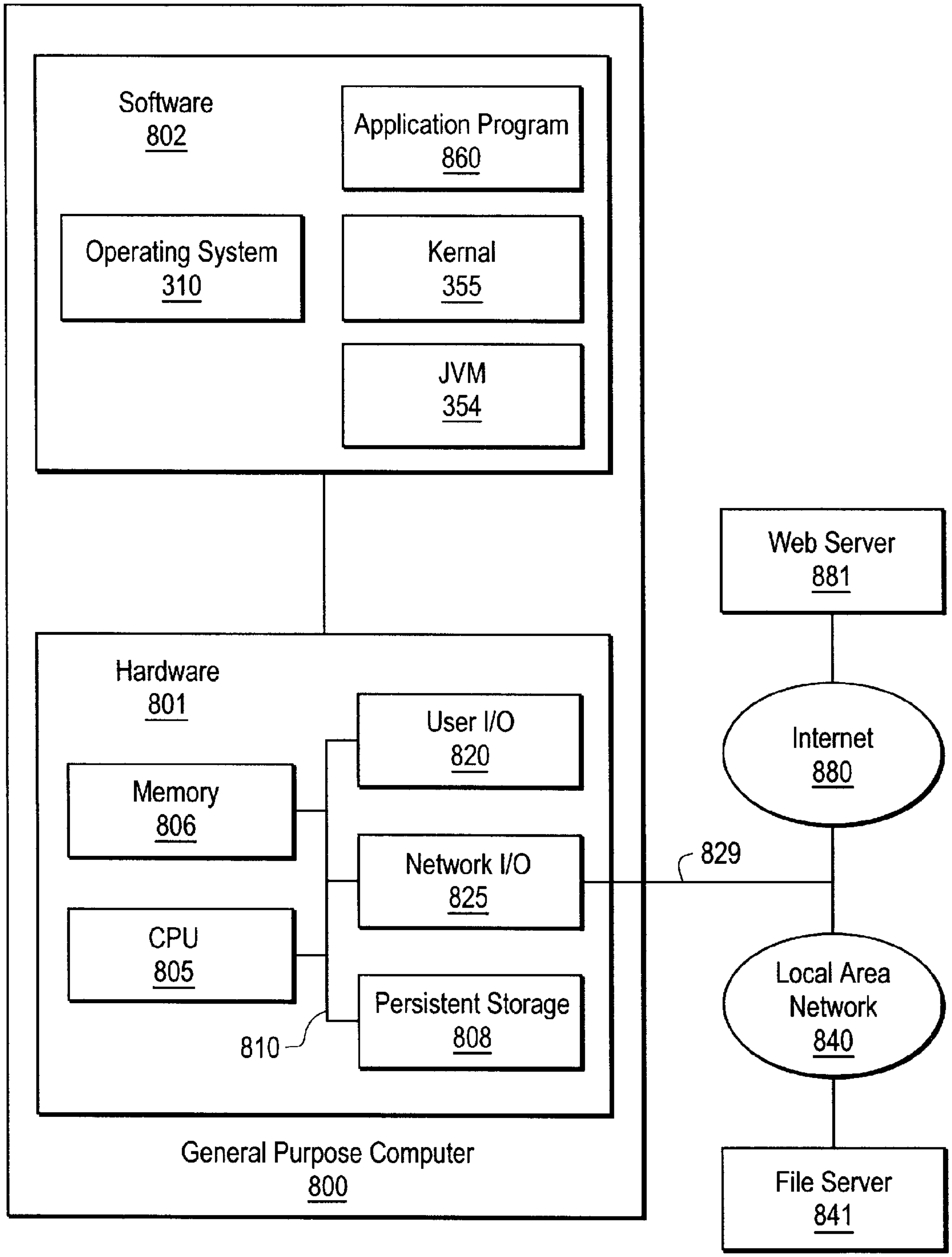

Software
802
Application Program
860
Operating System
310
Kernal
355
JVM
354
Web Server
881
Hardware
801
User I/O
820
Internet
880
Memory
806
Network I/O
825
829
CPU
805
Local Area
Network
840
Persistent Storage
808
810
General Purpose Computer
800
File Server
841

FIG. 8

SMART STUB OR ENTERPRISE JAVA^{TM} BEAN IN A DISTRIBUTED PROCESSING SYSTEM

CROSS REFERENCE TO RELATED APPLICATIONS

This application claims the benefit of U.S. Provisional Application No. 60/107,167, filed Nov. 5, 1998.

The following copending U.S. patent applications are assigned to the assignee of the present application, and their disclosures are incorporated herein by reference:

(A) Ser. No. 09/405,318 filed Sep. 23, 1999 by Dean B. Jacobs and Anno R. Langen, and originally entitled, "CLUSTERED ENTERPRISE JAVA™ HAVING A MESSAGE PASSING KERNEL IN A DISTRIBUTED PROCESSING SYSTEM";

(B) Ser. No. 09/405,508 filed Sep. 23, 1999 by Dean B. Jacobs and Eric M. Halpern, and entitled, "A DUPLICATED NAMING SERVICE IN A DISTRIBUTED PROCESSING SYSTEM"; and

(C) Ser. No. 09/405,500 filed Sep. 23, 1999 by Dean B. Jacobs and Anno R. Langen, and originally entitled, "CLUSTERED ENTERPRISE JAVA™ IN A SECURE DISTRIBUTED PROCESSING SYSTEM".

FIELD OF THE INVENTION

The present invention relates to distributed processing systems and, in particular, computer software in distributed processing systems.

BACKGROUND OF THE INVENTION

There are several types of distributed processing systems. Generally, a distributed processing system includes a plurality of processing devices, such as two computers coupled to a communication medium. Communication mediums may include wired mediums, wireless mediums, or combinations thereof, such as an Ethernet local area network or a cellular network. In a distributed processing system, at least one processing device may transfer information on the communication medium to another processing device.

Client/server architecture **110** illustrated in FIG. **1***a* is one type of distributed processing system. Client/server architecture **110** includes at least two processing devices, illustrated as client **105** and application server **103**. Additional clients may also be coupled to communication medium **104**, such as client **108**.

Typically, server **103** hosts business logic and/or coordinates transactions in providing a service to another processing device, such as client **103** and/or client **108**. Application server **103** is typically programmed with software for providing a service. The software may be programmed using a variety of programming models, such as Enterprise Java™ Bean ("EJB") **100***b* as illustrated in FIGS. **1***a*–*b*. The service may include, for example, retrieving and transferring data from a database, providing an image and/or calculating an equation. For example, server **103** may retrieve data from database **101***a* in persistent storage **101** over communication medium **102** in response to a request from client **105**. Application server **103** then may transfer the requested data over communication medium **104** to client **105**.

A client is a processing device which utilizes a service from a server and may request the service. Often a user **106** interacts with client **106** and may cause client **105** to request service over a communication medium **104** from application server **103**. A client often handles direct interactions with end users, such as accepting requests and displaying results.

A variety of different types of software may be used to program application server **103** and/or client **105**. One programming language is the Java™ programming language. Java™ application object code is loaded into a Java™ virtual machine ("JVM"). A JVM is a program loaded onto a processing device which emulates a particular machine or processing device. More information on the Java™ programming language may be obtained at http://www.javasoft.com, which is incorporated by reference herein.

FIG. **1***b* illustrates several Java™ Enterprise Application Programming Interfaces ("APIs") **100** that allow Java™ application code to remain independent from underlying transaction systems, data-bases and network infrastructure. Java™ Enterprise APIs **100** include, for example, remote method invocation ("RMI") **100***a*, EJBs **100***b*, and Java™ Naming and Directory Interface (JNDI) **100***c*.

RMI **100***a* is a distributed programming model often used in peer-to-peer architecture described below. In particular, a set of classes and interfaces enables one Java™ object to call the public method of another Java™ object running on a different JVM.

An instance of EJB **100***b* is typically used in a client/server architecture described above. An instance of EJB **100***b* is a software component or a reusable pre-built piece of encapsulated application code that can be combined with other components. Typically, an instance of EJB **100***b* contains business logic. An EJB **100***b* instance stored on server **103** typically manages persistence, transactions, concurrency, threading, and security.

JNDI **100***c* provides directory and naming functions to Java™ software applications.

Client/server architecture **110** has many disadvantages. First, architecture **110** does not scale well because server **103** has to handle many connections. In other words, the number of clients which may be added to server **103** is limited. In addition, adding twice as many processing devices (clients) does not necessarily provide you with twice as much performance. Second, it is difficult to maintain application code on clients **105** and **108**. Third, architecture **110** is susceptible to system failures or a single point of failure. If server **101** fails and a backup is not available, client **105** will not be able to obtain the service.

FIG. **1***c* illustrates a multi-tier architecture **160**. Clients **151, 152** manage direct interactions with end users, accepting requests and display results. Application server **153** hosts the application code, coordinates communications, synchronizations, and transactions. Database server **154** and portable storage device **155** provides durable transactional management of the data.

Multi-tier architecture **160** has similar client/server architecture **110** disadvantages described above.

FIG. 2 illustrates peer-to-peer architecture **214**. Processing devices **216, 217** and **218** are coupled to communication medium **213**. Processing devices **216, 217**, and **218** include network software **210***a*, **210***b*, and **210***c* for communicating over medium **213**. Typically, each processing device in a peer-to-peer architecture has similar processing capabilities and applications. Examples of peer-to-peer program models include Common Object Request Broker Architecture ("CORBA") and Distributed Object Component Model ("DCOM") architecture.

In a platform specific distributed processing system, each processing device may run the same operating system. This allows the use of proprietary hardware, such as shared disks, multi-tailed disks, and high speed interconnects, for communicating between processing devices. Examples of platform-specific distributed processing systems include IBM® Corporation's S/390® Parallel Sysplex®, Compaq's Tandem Division Himalaya servers, Compaq's Digital Equipment Corporation™ (DEC™) Division OpenVMS™ Cluster software, and Microsoft® Corporation Windows NT® cluster services (Wolfpack).

FIG. 2*b* illustrates a transaction processing (TP) architecture **220**. In particular, TP architecture **220** illustrates a BEA® Systems, Inc. TUXEDO® architecture. TP monitor **224** is coupled to processing devices ATM **221**, PC **222**, and TP monitor **223** by communication medium **280**, **281**, and **282**, respectively. ATM **221** may be an automated teller machine, PC **222** may be a personal computer, and TP monitor **223** may be another transaction processor monitor. TP monitor **224** is coupled to back-end servers **225**, **226**, and **227** by communication mediums **283**, **284**, and **285**. Server **225** is coupled to persistent storage device **287**, storing database **289**, by communication medium **286**. TP monitor **224** includes a workflow controller **224*a*** for routing service requests from processing devices, such as ATM **221**, PC **222**, or TP monitor **223**, to various servers such as server **225**, **226** and **227**. Work flow controller **224*a*** enables (1) workload balancing between servers, (2) limited scalability or allowing for additional servers and/or clients, (3) fault tolerance of redundant backend servers (or a service request may be sent by a workflow controller to a server which has not failed), and (4) session concentration to limit the number of simultaneous connections to back-end servers. Examples of other transaction processing architectures include IBM® Corporation's CICS®, Compaq's Tandem Division Pathway/Ford/TS, Compaq's DEC™ ACMS, and Transarc Corporation's Encina.

TP architecture **220** also has many disadvantages. First, a failure of a single processing device or TP monitor **224** may render the network inoperable. Second, the scalability or number of processing devices (both servers and clients) coupled to TP monitor **224** may be limited by TP monitor **224** hardware and software. Third, flexibility in routing a client request to a server is limited. For example, if communication medium **280** is inoperable, but communication medium **290** becomes available, ATM **221** typically may not request service directly from server **225** over communication medium **290** and must access TP monitor **224**. Fourth, a client typically does not know the state of a back-end server or other processing device. Fifth, no industry standard software or APIs are used for load balancing. And sixth, a client typically may not select a particular server even if the client has relevant information which would enable efficient service.

Therefore, it is desirable to provide a distributed processing system and, in particular, distributed processing system software that has the advantages of the prior art distributed processing systems without the inherent disadvantages. The software should allow for industry standard APIs which are typically used in either client/server, multi-tier, or peer-to-peer distributed processing systems. The software should support a variety of computer programming models. Further, the software should enable (1) enhanced fault tolerance, (2) efficient scalability, (3) effective load balancing, and (4) session concentration control. The improved computer software should allow for rerouting or network reconfiguration. Also, the computer software should allow for the determination of the state of a processing device.

SUMMARY OF THE INVENTION

An improved distributed processing system is provided and, in particular, computer software for a distributed processing system is provided. The computer software improves the fault tolerance of the distributed processing system as well as enables efficient scalability. The computer software allows for efficient load balancing and session concentration. The computer software supports rerouting or reconfiguration of a computer network. The computer software supports a variety of computer programming models and allows for the use of industry standard APIs that are used in both client/server and peer-to-peer distributed processing architectures. The computer software enables a determination of the state of a server or other processing device. The computer software also supports message forwarding under a variety of circumstances, including a security model.

According to one aspect of the present invention, a distributed processing system comprises a communication medium coupled to a first processing device and a second processing device. The first processing device includes a first software program emulating a processing device ("JVM1") including a first kernel software layer having a data structure ("RJVM1"). The second processing device includes a first software program emulating a processing device ("JVM2") including a first kernel software layer having a data structure ("RJVM2"). A message from the first processing device is transferred to the second processing device through the first kernel software layer and the first software program in the first processing device to the first kernel software layer and the first software program in the second processing device.

According to another aspect of the present invention, the first software program in the first processing device is a Java™ virtual machine ("JVM") and the data structure in the first processing device is a remote Java™ virtual machine ("RJVM"). Similarly, the first software program in the second processing device is a JVM and the data structure in the second processing device is a RJVM. The RJVM in the second processing device corresponds to the JVM in the first processing device.

According to another aspect of the present invention, the RJVM in the first processing device includes a socket manager software component, a thread manager software component, a message routing software component, a message compression software component, and/or a peer-gone detection software component.

According to another aspect of the present invention, the first processing device communicates with the second processing device using a protocol selected from the group consisting of Transmission Control Protocol ("TCP"), Secure Sockets Layer ("SSL"), Hypertext Transport Protocol ("HTTP") tunneling, and Internet InterORB Protocol ("IIOP") tunneling.

According to another aspect of the present invention, the first processing device includes memory storage for a Java™ application.

According to another aspect of the present invention, the first processing device is a peer of the second processing device. Also, the first processing device is a server and the second processing device is a client.

According to another aspect of the present invention, a second communication medium is coupled to the second processing device. A third processing device is coupled to the second communication medium. The third processing device includes a first software program emulating a processing device ("JVM3"), including a kernel software layer having a first data structure ("RJVM1"), and a second data structure ("RJVM2").

According to still another aspect of the present invention, the first processing device includes a stub having a replicahandler software component. The replica-handler software component includes a load balancing software component and a failover software component.

According to another aspect of the present invention, the first processing device includes an Enterprise Java™ Bean object.

According to still another aspect of the present invention, the first processing device includes a naming tree having a pool of stubs stored at a node of the tree and the second processing device includes a duplicate of the naming tree.

According to still another aspect of the present invention, the first processing device includes an application program coded in a stateless program model and the application program includes a stateless session bean.

According to still another aspect of the present invention, the first processing device includes an application program coded in a stateless factory program model and the application program includes a stateful session bean.

According to still another aspect of the present invention, the first processing device includes an application program coded in a stateful program model and the application program includes an entity session bean.

According to still another aspect of the present invention, an article of manufacture including an information storage medium is provided. The article of manufacture comprises a first set of digital information for transferring a message from a RJVM in a first processing device to a RJVM in a second processing device.

According to another aspect of the present invention, the article of manufacture comprises a first set of digital information, including a stub having a load balancing software program for selecting a service provider from a plurality of service providers.

According to another aspect of the present invention, the stub has a failover software component for removing a failed service provider from the plurality of service providers.

According to another aspect of the present invention, the load balancing software component selects a service provider based on an affinity for a particular service provider.

According to another aspect of the present invention, the load balancing software component selects a service provider in a round robin manner.

According to another aspect of the present invention, the load balancing software component randomly selects a service provider.

According to another aspect of the present invention, the load balancing software component selects a service provider from the plurality of service providers based upon the load of each service provider.

According to another aspect of the present invention, the load balancing software component selects a service provider from the plurality of service providers based upon the data type requested.

According to another aspect of the present invention, the load balancing software component selects a service provider from the plurality of service providers based upon the closest physical service provider.

According to another aspect of the present invention, the load balancing software component selects a service provider from the plurality of service providers based upon a time period in which each service provider responds.

According to another aspect of the present invention, the article of manufacture comprises a first set of digital information, including an Enterprise Java™ Bean object for selecting a service provider from a plurality of service providers.

According to another aspect of the present invention, a stub is stored in a processing device in a distributed processing system. The stub includes a method comprising the steps of obtaining a list of service providers and selecting a service provider from the list of service providers.

According to another aspect of the present invention, the method further includes removing a failed service provider from the list of service providers.

According to still another aspect of the present invention, an apparatus comprises a communication medium coupled to a first processing device and a second processing device. The first processing device stores a naming tree including a remote method invocation ("RMI") stub for accessing a service provider. The second processing device has a duplicate naming tree and the service provider.

According to another aspect of the present invention, the naming tree has a node including a service pool of current service providers.

According to another aspect of the present invention, the service pool includes a stub.

According to another aspect of the present invention, a distributed processing system comprises a first computer coupled to a second computer. The first computer has a naming tree, including a remote invocation stub for accessing a service provider. The second computer has a replicated naming tree and the service provider.

According to another aspect of the present invention, a distributed processing system comprising a first processing device coupled to a second processing device is provided. The first processing device has a JVM and a first kernel software layer including a first RJVM. The second processing device includes a first JVM and a first kernel software layer including a second RJVM. A message may be transferred from the first processing device to the second processing device when there is not a socket available between the first JVM and the second JVM.

According to another aspect of the present invention, the first processing device is running under an applet security model, behind a firewall or is a client, and the second processing device is also a client.

Other aspects and advantages of the present invention can be seen upon review of the figures, the detailed description, and the claims which follow.

BRIEF DESCRIPTION OF THE FIGURES

FIG. **1***a* illustrates a prior art client/server architecture;

FIG. **1***b* illustrates a prior art Java™ enterprise APIs;

FIG. **1***c* illustrates a multi-tier architecture;

FIG. **2***a* illustrates a prior art peer-to-peer architecture;

FIG. **2***b* illustrates a prior art transaction processing architecture;

FIG. **3***a* illustrates a simplified software block diagram of an embodiment of the present invention;

FIG. **3***b* illustrates a simplified software block diagram of the kernel illustrated in FIG. **3***a;*

FIG. **3***c* illustrates a clustered enterprise Java™ architecture;

FIG. **4** illustrates a clustered enterprise Java™ naming service architecture;

FIG. **5***a* illustrates a smart stub architecture;

FIG. **5***b* illustrates an EJB object architecture;

FIG. **6***a* is a control flow chart illustrating a load balancing method;

FIGS. **6***b–g* are control flow charts illustrating load balancing methods;

FIG. **7** is a control flow chart illustrating a failover method;

FIG. **8** illustrates hardware and software components of a client/server in the clustered enterprise Java™ architecture shown in FIGS. **3–5**.

The invention will be better understood with reference to the drawings and detailed description below. In the drawings, like reference numerals indicate like components.

DETAILED DESCRIPTION

I. Clustered Enterprise Java™ Distributed Processing System

A. Clustered Enterprise Java™ Software Architecture

FIG. **3***a* illustrates a simplified block diagram **380** of the software layers in a processing device of a clustered enterprise Java™ system, according to an embodiment of the present invention. A detailed description of a clustered enterprise Java™ distributed processing system is described below. The first layer of software includes a communication medium software driver **351** for transferring and receiving information on a communication medium, such as an ethernet local area network. An operating system **310** including a transmission control protocol ("TCP") software component **353** and internet protocol ("IP") software component **352** are upper software layers for retrieving and sending packages or blocks of information in a particular format. An "upper" software layer is generally defined as a software component which utilizes or accesses one or more "lower" software layers or software components. A JVM **354** is then implemented. A kernel **355** having a remote Java™ virtual machine **356** is then layered on JVM **354**. Kernel **355**. described in detail below, is used to transfer messages between processing devices in a clustered enterprise Java™ distributed processing system. Remote method invocation **357** and enterprise Java™ bean **358** are upper software layers of kernel **355**. EJB **358** is a container for a variety of Java™ applications.

FIG. **3***b* illustrates a detailed view of kernel **355** illustrated in FIG. **3***a*. Kernel **355** includes a socket manager component **363**, thread manager **364** component, and RJVM **356**. RJVM **356** is a data structure including message routing software component **360**, message compression software component **361** including abbreviation table **161***c*, and peer-gone detection software component **362**. RJVM **356** and thread manager component **364** interact with socket manager component **363** to transfer information between processing devices.

B. Distributed Processing System

FIG. **3** illustrates a simplified block diagram of a clustered enterprise Java™ distributed processing system **300**. Processing devices are coupled to communication medium **301**. Communication medium **301** may be a wired and/or wireless communication medium or combination thereof. In an embodiment, communication medium **301** is a local-area-network (LAN). In an alternate embodiment, communication medium **301** is a world-area-network (WAN) such as the Internet or World Wide Web. In still another embodiment, communication medium **301** is both a LAN and a WAN.

A variety of different types of processing devices may be coupled to communication medium **301**. In an embodiment, a processing device may be a general purpose computer **100** as illustrated in FIG. **8** and described below. One of ordinary skill in the art would understand that FIG. **8** and the below description describes one particular type of processing device where multiple other types of processing devices with a different software and hardware configurations could be utilized in accordance with an embodiment of the present invention. In an alternate embodiment, a processing device may be a printer, handheld computer, laptop computer, scanner, cellular telephone, pager, or equivalent thereof.

FIG. **3***c* illustrates an embodiment of the present invention in which servers **302** and **303** are coupled to communication medium **301**. Server **303** is also coupled to communication medium **305** which may have similar embodiments as described above in regard to communication medium **301**. Client **304** is also coupled to communication medium **305**. In an alternate embodiment, client **304** may be coupled to communication medium **301** as illustrated by the dashed line and box in FIG. **3***c*. It should be understood that in alternate embodiments, server **302** is (1) both a client and a server, or (2) a client. Similarly, FIG. **3** illustrates an embodiment in which three processing devices are shown wherein other embodiments of the present invention include multiple other processing devices or communication mediums as illustrated by the ellipses.

Server **302** transfers information over communication medium **301** to server **303** by using network software **302***a* and network software **303***a*, respectively. In an embodiment, network software **302***a*, **303***a*, and **304***a* include communication medium software driver **351**, Transmission Control Protocol software **353**, and Internet Protocol software **352** ("TCP/IP"). Client **304** also includes network software **304***a* for transferring information to server **303** over communication medium **305**. Network software **303***a* in server **303** is also used to transfer information to client **304** by way of communication medium **305**.

According to an embodiment of the present invention, each processing device in clustered enterprise Java™ architecture **300** includes a message-passing kernel **355** that supports both multi-tier and peer-to-peer functionality. A kernel is a software program used to provide fundamental services to other software programs on a processing device.

In particular, server **302**, server **303**, and client **304** have kernels **302***b*, **303***b*, and **304***b*, respectively. In particular, in order for two JVMs to interact, whether they are clients or servers, each JVM constructs an RJVM representing the other. Messages are sent from the upper layer on one side, through a corresponding RJVM, across the communication medium, through the peer RJVM, and delivered to the upper layer on the other side. In various embodiments, messages can be transferred using a variety of different protocols, including, but not limited to, Transmission Control Protocol/Internet Protocol ("TCP/IP"), Secure Sockets Layer ("SSL"), Hypertext Transport Protocol ("HTTP") tunneling, and Internet InterORB Protocol ("IIOP") tunneling, and combinations thereof. The RJVMs and socket managers create and maintain the sockets underlying these protocols and share them between all objects in the upper layers. A socket is a logical location representing a terminal between processing devices in a distributed processing system. The kernel maintains a pool of execute threads and thread manager software component **364** multiplexes the threads between socket reading and request execution. A thread is a sequence of executing program code segments or functions.

For example, server **302** includes JVM1 and Java™ application **302***c*. Server **302** also includes a RJVM2 representing the JVM2 of server **303**. If a message is to be sent from server **302** to server **303**, the message is sent through RJVM2 in server **302** to RJVM1 in server **303**.

C. Message Forwarding

Clustered enterprise Java™ network **300** is able to forward a message through an intermediate server. This functionality is important if a client requests a service from a back-end server through a front-end gateway. For example, a message from server **302** (client **302**) and, in particular, JVM**1** may be forwarded to client **304** (back-end server **304**) or JVM**3** through server **303** (front-end gateway) or JVM**2**. This functionality is important in controlling session concentration or how many connections are established between a server and various clients.

Further, message forwarding may be used in circumstances where a socket cannot be created between two JVMs. For example, a sender of a message is running under the applet security model which does not allow for a socket to be created to the original server. A detailed description of the applet security model is provided at http//:www.javasoft.com, which is incorporated herein by reference. Another example includes when the receiver of the message is behind a firewall. Also, as described below, message forwarding is applicable if the sender is a client and the receiver is a client and thus does not accept incoming sockets.

For example, if a message is sent from server **302** to client **304**, the message would have to be routed through server **303**. In particular, a message handoff, as illustrated by **302***f*, between RJVM**3** (representing client **304**) would be made to RJVM**2** (representing server **303**) in server **302**. The message would be transferred using sockets **302***e* between RJVM**2** in server **302** and RJVM**1** in server **303**. The message would then be handed off, as illustrated by dashed line **303***f*, from RJVM**1** to RJVM**3** in server **303**. The message would then be passed between sockets of RJVM**3** in server **303** and RJVM**2** in client **304**. The message then would be passed, as illustrated by the dashed line **304***f*, from RJVM**2** in client **304** to RJVM**1** in client **304**.

D. Rerouting

An RJVM in client/server is able to switch communication paths or communication mediums to other RJVMs at any time. For example, if client **304** creates a direct socket to server **302**, server **302** is able to start using the socket instead of message forwarding through server **303**. This embodiment is illustrated by a dashed line and box representing client **304**. In an embodiment, the use of transferring messages by RJVMs ensures reliable, in-order message delivery after the occurrence of a network reconfiguration. For example, if client **304** was reconfigured to communication medium **301** instead of communication medium **305** as illustrated in FIG. **3**. In an alternate embodiment, messages may not be delivered in order.

An RJVM performs several end-to-end operations that are carried through routing. First, an RJVM is responsible for detecting when a respective client/server has unexpectedly died. In an embodiment, peer-gone selection software component **362**, as illustrated in FIG. **3***b*, is responsible for this function. In an embodiment, an RJVM sends a heartbeat message to other clients/servers when no other message has been sent in a predetermined time period. If the client/server does not receive a heartbeat message in the predetermined count time, a failed client/server which should have sent the heartbeat, is detected. In an embodiment, a failed client/server is detected by connection timeouts or if no messages have been sent by the failed client/server in a predetermined amount of time. In still another embodiment, a failed socket indicates a failed server/client.

Second, during message serialization, RJVMs, in particular, message compression software **360**, abbreviate commonly transmitted data values to reduce message size. To accomplish this, each JVM/RJVM pair maintains matching abbreviation tables. For example, JVM**1** includes an abbreviation table and RJVM**1** includes a matching abbreviation table. During message forwarding between an intermediate server, the body of a message is not deserialized on the intermediate server in route.

E. Multi-tier/Peer-to-Peer Functionality

Clustered enterprise Java™ architecture **300** allows for multitier and peer-to-peer programming.

Clustered enterprise Java™ architecture **300** supports an explicit syntax for client/server programming consistent with a multitier distributed processing architecture. As an example, the following client-side code fragment writes an informational message to a server's log file:

```
T3Client clnt=new T3Client(“t3://acme:7001”);
LogServices log=clnt.getT3Services( ).log( );
log.info(“Hello from a client”);
```

The first line establishes a session with the acme server using the t**3** protocol. If RJVMs do not already exist, each JVM constructs an RJVM for the other and an underlying TCP socket is established. The client-side representation of this session—the T3Client object—and the server-side representation communicate through these RJVMs. The server-side supports a variety of services, including database access, remote file access, workspaces, events, and logging. The second line obtains a LogServices object and the third line writes the message.

Clustered enterprise Java™ computer architecture **300** also supports a server-neutral syntax consistent with a peer-to-peer distributed processing architecture. As an example, the following code fragment obtains a stub for an RMI object from the JNDI-compliant naming service on a server and invokes one of its methods.

```
Hashtable env=new Hashtable( );
env.put(Context.PROVIDER_URL, “t3://acme:7001”);
env.put(Context.INITIAL_CONTEXT_FACTORY,
“weblogic.jndi.WebLogicinitialContextFactory”);
Context ctx=new InitialContext(env);
Example e=(Example) ctx.lookup(“acme.eng.example”);
result=e.example(37);
```

In an embodiment, JNDI naming contexts are packaged as RMI objects to implement remote access. Thus, the above code illustrates a kind of RMI bootstrapping. The first four lines obtain an RMI stub for the initial context on the acme server. If RJVMs do not already exist, each side constructs an RJVM for the other and an underlying TCP socket for the t**3** protocol is established. The caller-side object—the RMI stub—and the callee-side object—an RMI impl-communicate through the RJVMs. The fifth line looks up another RMI object, an Example, at the name acme.eng.example and the sixth line invokes one of the Example methods. In an embodiment, the Example impl is not on the same processing device as the naming service. In another embodiment, the Example impl is on a client. Invocation of the Example object leads to the creation of the appropriate RJVMs if they do not already exist.

II. Replica-Aware or Smart Stubs/EJB Objects

In FIG. **3***c*, a processing device is able to provide a service to other processing devices in architecture **300** by replicating RMI and/or EJB objects. Thus, architecture **300** is easily scalable and fault tolerant. An additional service may easily be added to architecture **300** by adding replicated RMI and/or EJB objects to an existing processing device or newly added processing device. Moreover, because the RMI and/or

EJB objects can be replicated throughout architecture **300**, a single processing device, multiple processing devices, and/or a communication medium may fail and still not render architecture **300** inoperable or significantly degraded.

FIG. **5***a* illustrates a replica-aware ("RA") or Smart stub **580** in architecture **500**. Architecture **500** includes client **504** coupled to communication medium **501**. Servers **502** and **503** are coupled to communication medium **501**, respectively. Persistent storage device **509** is coupled to server **502** and **503** by communication medium **560** and **561**, respectively. In various embodiments, communication medium **501**, **560**, and **561** may be wired and/or wireless communication mediums as described above. Similarly, in an embodiment, client **504**, server **502**, and server **503** may be both clients and servers as described above. One of ordinary skill in the art would understand that in alternate embodiments, multiple other servers and clients may be included in architecture **500** as illustrated by ellipses. Also, as stated above, in alternate embodiments, the hardware and software configuration of client **504**, server **502** and server **503** is described below and illustrated in FIG. **8**.

RA RMI stub **580** is a Smart stub which is able to find out about all of the service providers and switch between them based on a load balancing method **507** and/or failover method **508**. In an embodiment, an RA stub **580** includes a replica handler **506** that selects an appropriate load balancing method **507** and/or failover method **507**. In an alternate embodiment, a single load balancing method and/or single failover method is implemented. In alternate embodiments, replica handler **506** may include multiple load balancing methods and/or multiple failover methods and combinations thereof. In an embodiment, a replica handler **506** implements the following interface:

```
public interface ReplicaHandler {
   Object loadBalance(Object currentProvider) throws
RefreshAbortedException;
   Object failOver(Object failedProvider,
   RemoteException e) throws
RemoteException;
}
```

Immediately before invoking a method, RA stub **580** calls load balance method **507**, which takes the current server and returns a replacement. For example, client **504** may be using server **502** for retrieving data for database **509***a* or personal storage device **509**. Load balance method **507** may switch to server **503** because server **502** is overloaded with service requests. Handler **506** may choose a server replacement entirely on the caller, perhaps using information about server **502** load, or handler **506** may request server **502** for retrieving a particular type of data. For example, handler **506** may select a particular server for calculating an equation because the server has enhanced calculation capability. In an embodiment, replica handler **506** need not actually switch providers on every invocation because replica handler **506** is trying to minimize the number of connections that are created.

FIG. **6***a* is a control flow diagram illustrating the load balancing software **507** illustrated in FIGS. **5***a*–*b*. It should be understood that FIG. **6***a* is a control flow diagram illustrating the logical sequence of functions or steps which are completed by software in load balancing method **507**. In alternate embodiments, additional functions or steps are completed. Further, in an alternate embodiment, hardware may perform a particular function or all the functions.

Load balancing software **507** begins as indicated by circle **600**. A determination is then made in logic block **601** as to whether the calling thread established "an affinity" for a particular server. A client has an affinity for the server that coordinates its current transaction and a server has an affinity for itself. If an affinity is established, control is passed to logic block **602**, otherwise control is passed to logic block **604**. A determination is made in logic block **602** whether the affinity server provides the service requested. If so, control is passed to logic block **603**. Otherwise, control is passed to logic block **604**. The provider of the service on the affinity server is returned to the client in logic block **603**. In logic block **604**, a naming service is contacted and an updated list of the current service providers is obtained. A getNextProvider method is called to obtain a service provider in logic block **605**. Various embodiments of the getNextProvider method are illustrated in FIGS. **6***b*–*g* and described in detail below. The service is obtained in logic block **606**. Failover method **508** is then called if service is not provided in logic block **606** and load balancing method **507** exits as illustrated by logic block **608**. An embodiment of failover method **508** is illustrated in FIG. **7** and described in detail below.

FIGS. **6***b*–*g* illustrate various embodiments of a getNextProvider method used in logic block **605** of FIG. **6***a*. As illustrated in FIG. **6***b*, the getNextProvider method selects a service provider in a round robin manner. A getNextProvider method **620** is entered as illustrated by circle **621**. A list of current service providers is obtained in logic block **622**. A pointer is incremented in logic block **623**. The next service provider is selected based upon the pointer in logic block **624** and the new service provider is returned in logic block **625** and getNextProvider method **620** exits as illustrated by circle **626**.

FIG. **6***c* illustrates an alternate embodiment of a getNextProvider method which obtains a service provider by selecting a service provider randomly. A getNextProvider method **630** is entered as illustrated by circle **631**. A list of current service providers is obtained as illustrated by logic block **632**. The next service provider is selected randomly as illustrated by logic block **633** and a new service provider is returned in logic block **634**. The getNextProvider method **630** then exits, as illustrated by circle **635**.

Still another embodiment of a getNextProvider method is illustrated in FIG. **6***d* which obtains a service provider based upon the load of the service providers. A getNextProvider method **640** is entered as illustrated by circle **641**. A list of current service providers is obtained in logic block **642**. The load of each service provider is obtained in logic block **643**. The service provider with the least load is then selected in logic block **644**. The new service provider is then returned in logic block **645** and getNextProvider method **640** exits as illustrated by circle **646**.

An alternate embodiment of a getNextProvider method is illustrated in FIG. **6***e* which obtains a service provider based upon the type of data obtained from the service provider. A getNextProvider method **650** is entered as illustrated by circle **651**. A list of current service providers is obtained in logic block **652**. The type of data requested from the service providers is determined in logic block **653**. The service provider is then selected based on the data type in logic block **654**. The service provider is returned in logic block **655** and getNextProvider method **650** exits as illustrated by circle **656**.

Still another embodiment of a getNextProvider method is illustrated in FIG. **6***f* which selects a service provider based upon the physical location of the service providers. A getNextProvider method **660** is entered as illustrated by circle **661**. A list of service providers is obtained as illustrated by logic block **662**. The physical distance to each service provider is determined in logic block **663** and the service provider which has the closest physical distance to the requesting client is selected in logic block **664**. The new service provider is then returned in logic block **665** and the getNextProvider method **660** exits as illustrated by circle **666**.

Still a further embodiment of the getNextProvider method is illustrated in FIG. **6***g* and selects a service provider based on the amount of time taken for the service provider to respond to previous requests. Control of getNextProvider method **670** is entered as illustrated by circle **671**. A list of current service providers is obtained in logic block **672**. The time period for each service provider to respond to a particular message is determined in logic block **673**. The service provider which responds in the shortest time period is selected in logic block **674**. The new service provider is then returned in logic block **675** and control from getNextProvider method **670** exits as illustrated by circle **676**.

If invocation of a service method fails in such a way that a retry is warranted, RA **580** stub calls failover method **508**, which takes the failed server and an exception indicating what the failure was and returns a new server for the retry. If a new server is unavailable, RA stub **580** throws an exception.

FIG. **7** is a control flow chart illustrating failover software **508** shown in FIGS. **5***a–b*. Failover method **508** is entered as illustrated by circle **700**. A failed provider from the list of current providers of services is removed in logic block **701**. A getNextProvider method is then called in order to obtain a service provider. The new service provider is then returned in logic block **703** and failover method **508** exits as illustrated by circle **704**.

While FIGS. **6**–**7** illustrate embodiments of a replica handler **506**, alternate embodiments include the following functions or combinations thereof implemented in a round robin manner.

First, a list of servers or service providers of a service is maintained. Whenever the list needs to be used and the list has not been recently updated, handler **506** contacts a naming service as described below and obtains an up-to-date list of providers.

Second, if handler **506** is about to select a provider from the list and there is an existing RJVM-level connection to the hosting server over which no messages have been received during the last heartbeat period, handler **506** skips that provider. In an embodiment, a server may later recover since death of peer is determined after several such heartbeat periods. Thus, load balancing on the basis of server load is obtained.

Third, when a provider fails, handler **506** removes the provider from the list. This avoids delays caused by repeated attempts to use non-working service providers.

Fourth, if a service is being invoked from a server that hosts a provider of the service, then that provider is used. This facilitates co-location of providers for chained invokes of services.

Fifth, if a service is being invoked within the scope of a transaction and the server acting as transaction coordinator hosts a provider of the service, then that provider is used. This facilitates co-location of providers within a transaction.

The failures that can occur during a method invocation may be classified as being either (1) application-related, or (2) infrastructure-related. RA stub **580** will not retry an operation in the event of an application-related failure, since there can be no expectation that matters will improve. In the event of an infrastructure-related failure, RA stub **580** may or may not be able to safely retry the operation. Some initial non-idempotent operation, such as incrementing the value of a field in a database, might have completed. In an embodiment, RA stub **580** will retry after an infrastructure failure only if either (1) the user has declared that the service methods are idempotent, or (2) the system can determine that processing of the request never started. As an example of the latter, RA stub **580** will retry if, as part of load balancing method, stub **580** switches to a service provider whose host has failed. As another example, a RA stub **580** will retry if it gets a negative acknowledgment to a transactional operation.

A RMI compiler recognizes a special flag that instructs the compiler to generate an RA stub for an object. An additional flag can be used to specify that the service methods are idempotent. In an embodiment, RA stub **580** will use the replica handler described above and illustrated in FIG. **5***a*. An additional flag may be used to specify a different handler. In addition, at the point a service is deployed, i.e., bound into a clustered naming service as described below, the handler may be overridden.

FIG. **5***b* illustrates another embodiment of the present invention in which an EJB object **551** is used instead of a stub, as shown in FIG. **5***a*.

III. Replicated JNDI-compliant Naming Service

As illustrated in FIG. **4**, access to service providers in architecture **400** is obtained through a JNDI-compliant naming service, which is replicated across architecture **400** so there is no single point of failure. Accordingly, if a processing device which offers a JNDI-compliant naming service fails, another processing device having a replicated naming service is available. To offer an instance of a service, a server advertises a provider of the service at a particular node in a replicated naming tree. In an embodiment, each server adds a RA stub for the provider to a compatible service pool stored at the node in the server's copy of the naming tree. If the type of a new offer is incompatible with the type of offers in an existing pool, the new offer is made pending and a callback is made through a ConflictHandler interface. After either type of offer is retracted, the other will ultimately be installed everywhere. When a client looks up the service, the client obtains a RA stub that contacts the service pool to refresh the client's list of service providers.

FIG. **4** illustrates a replicated naming service in architecture **400**. In an embodiment, servers **302** and **303** offer an example service provider **P1** and **P2**, respectively, and has a replica of the naming service tree **402** and **403**, respectively. The node acme.eng.example in naming service tree **402** and **403** has a service pool **402***a* and **403***a*, respectively, containing a reference to Example service provider **P1** and **P2**. Client **304** obtains a RA stub **304***e* by doing a naming service lookup at the acme.eng.example node. Stub **304***e* contacts an instance of a service pool to obtain a current list of references to available service providers. Stub **304***e* may switch between the instances of a service pool as needed for load-balancing and failover.

Stubs for the initial context of the naming service are replica-aware or Smart stubs which initially load balance among naming service providers and switch in the event of a failure. Each instance of the naming service tree contains a complete list of the current naming service providers. The stub obtains a fresh list from the instance it is currently using. To bootstrap this process, the system uses Domain Naming Service ("DNS") to find a (potentially incomplete) initial list of instances and obtains the complete list from one of them. As an example, a stub for the initial context of the naming service can be obtained as follows:

```
Hashtable env=new Hashtable( );
env.put(Context.PROVIDER_URL, "t3://
   acmeCluster:7001");
```

```
env.put(Context.INITIAL_CONTEXT_FACTORY,
"weblogic.jndi.WebLogicInitialContextFactor");
Context ctx=new InitialContext(env);
```

Some subset of the servers in an architecture have been bound into DNS under the name acmeCluster. Moreover, an application is still able to specify the address of an individual server, but the application will then have a single point of failure when the application first attempts to obtain a stub.

A reliable multicast protocol is desirable. In an embodiment, provider stubs are distributed and replicated naming trees are created by an IP multicast or point-to-point protocol. In an IP multicast embodiment, there are three kinds of messages: Heartbeats, Announcements, and StateDumps. Heartbeats are used to carry information between servers and, by their absence, to identify failed servers. An Announcement contains a set of offers and retractions of services. The Announcements from each server are sequentially numbered. Each receiver processes an Announcement in order to identify lost Announcements. Each server includes in its Heartbeats the sequence number of the last Announcement it has sent. Negative Acknowledgments ("NAKs") for a lost Announcement are included in subsequent outgoing Heartbeats. To process NAKs, each server keeps a list of the last several Announcements that the server has sent. If a NAK arrives for an Announcement that has been deleted, the server sends a StateDump, which contains a complete list of the server's services and the sequence number of its next Announcement. When a new server joins an existing architecture, the new server NAKs for the first message from each other server, which results in StateDumps being sent. If a server does not receive a Heartbeat from another server after a predetermined period of time, the server retracts all services offered by the server not generating a Heartbeat.

IV. Programming Models

Applications used in the architecture illustrated in FIGS. **3**–**5** use one of three basic programming models: (1) stateless or direct, (2) stateless factory or indirect, or (3) stateful or targeted, depending on the way the application state is to be treated. In the stateless model, a Smart stub returned by a naming-service lookup directly references service providers.

```
Example e=(Example) ctx.lookup("acme.eng.example");
result1=e.example(37);
result2=e.example(38);
```

In this example, the two calls to example may be handled by different service providers since the Smart stub is able to switch between them in the interests of load balancing. Thus, the Example service object cannot internally store information on behalf of the application. Typically the stateless model is used only if the provider is stateless. As an example, a pure stateless provider might compute some mathematical function of its arguments and return the result. Stateless providers may store information on their own behalf, such as for accounting purposes. More importantly, stateless providers may access an underlying persistent storage device and load application state into memory on an as-needed basis. For example, in order for example to return the running sum of all values passed to it as arguments, example might read the previous sum from a database, add in its current argument, write the new value out, and then return it. This stateless service model promotes scalability.

In the stateless factory programming model, the Smart stub returned by the lookup is a factory that creates the desired service providers, which are not themselves Smart stubs.

```
ExampleFactory gf=(ExampleFactory)
ctx.lookup("acme.eng.example");
Example e=gf.create( );
result1=e.example(37);
result2=e.example(38);
```

In this example, the two calls to example are guaranteed to be handled by the same service provider. The service provider may therefore safely store information on behalf of the application. The stateless factory model should be used when the caller needs to engage in a "conversation" with the provider. For example, the caller and the provider might engage in a back-and-forth negotiation. Replica-aware stubs are generally the same in the stateless and stateless factory models, the only difference is whether the stubs refer to service providers or service provider factories.

A provider factory stub may failover at will in its effort to create a provider, since this operation is idempotent. To further increase the availability of an indirect service, application code must contain an explicit retry loop around the service creation and invocation.

```
while (true) {
  try {
    Example e=gf.create( );
    result1:=e.example(37);
    result2=e.example(38);
    break;
  } catch (Exception e) {
    if (!retryWarranted(e))
       throw e;
  }
}
```

This would, for example, handle the failure of a provider e that was successfully created by the factory. In this case, application code should determine whether non-idempotent operations completed. To further increase availability, application code might attempt to undo such operations and retry.

In the stateful programming model, a service provider is a long-lived, stateful object identified by some unique system-wide key. Examples of "entities" that might be accessed using this model include remote file systems and rows in a database table. A targeted provider may be accessed many times by many clients, unlike the other two models where each provider is used once by one client. Stubs for targeted providers can be obtained either by direct lookup, where the key is simply the naming-service name, or through a factory, where the key includes arguments to the create operation. In either case, the stub will not do load balancing or failover. Retries, if any, must explicitly obtain the stub again.

There are three kinds of beans in EJB, each of which maps to one of the three programming models. Stateless session beans are created on behalf of a particular caller, but maintain no internal state between calls. Stateless session beans map to the stateless model. Stateful session beans are created on behalf of a particular caller and maintain internal state between calls. Stateful session beans map to the stateless factory model. Entity beans are singular, stateful objects identified by a system-wide key. Entity beans map to the stateful model. All three types of beans are created by a factory called an EJB home. In an embodiment, both EJB homes and the beans they create are referenced using RMI. In an architecture as illustrated in FIGS. **3**–**5**, stubs for an EJB home are Smart stubs. Stubs for stateless session beans are Smart stubs, while stubs for stateful session beans and entity beans are not. The replica handler to use for an EJB-based service can be specified in its deployment descriptor.

To create an indirect RMI-based service, which is required if the object is to maintain state on behalf of the caller, the application code must explicitly construct the factory. A targeted RMI-based service can be created by running the RMI compiler without any special flags and then binding the resulting service into the replicated naming tree. A stub for the object will be bound directly into each instance of the naming tree and no service pool will be created. This provides a targeted service where the key is the naming-service name. In an embodiment, this is used to create remote file systems.

V. Hardware and Software Components

FIG. **8** shows hardware and software components of an exemplary server and/or client as illustrated in FIGS. **3–5**. The system of FIG. **8** includes a general-purpose computer **800** connected by one or more communication mediums, such as connection **829**, to a LAN **840** and also to a WAN, here illustrated as the Internet **880**. Through LAN **840**, computer **800** can communicate with other local computers, such as a file server **841**. In an embodiment, file server **801** is server **303** as illustrated in FIG. **3**. Through the Internet **880**, computer **800** can communicate with other computers, both local and remote, such as World Wide Web server **881**. In an embodiment, Web server **881** is server **303** as illustrated in FIG. **3**. As will be appreciated, the connection from computer **800** to Internet **880** can be made in various ways, e.g., directly via connection **829**, or through local-area network **840**, or by modem (not shown).

Computer **800** is a personal or office computer that can be, for example, a workstation, personal computer, or other single-user or multi-user computer system; an exemplary embodiment uses a Sun SPARC-20 workstation (Sun Microsystems, Inc., Mountain View, Calif.). For purposes of exposition, computer **800** can be conveniently divided into hardware components **801** and software components **802**; however, persons of ordinary skill in the art will appreciate that this division is conceptual and somewhat arbitrary, and that the line between hardware and software is not a hard and fast one. Further, it will be appreciated that the line between a host computer and its attached peripherals is not a hard and fast one, and that in particular, components that are considered peripherals of some computers are considered integral parts of other computers. Thus, for example, user I/O **820** can include a keyboard, a mouse, and a display monitor, each of which can be considered either a peripheral device or part of the computer itself, and can further include a local printer, which is typically considered to be a peripheral. As another example, persistent storage **808** can include a CD-ROM (compact disc read-only memory) unit, which can be either peripheral or built into the computer.

Hardware components **801** include a processor (CPU) **805**, memory **806**, persistent storage **808**, user I/O **820**, and network interface **825** which are coupled to bus **810**. These components are well understood by those of skill in the art and, accordingly, need be explained only briefly here.

Processor **805** can be, for example, a microprocessor or a collection of microprocessors configured for multiprocessing.

Memory **806** can include read-only memory (ROM), randomaccess memory (RAM), virtual memory, or other memory technologies, singly or in combination. Persistent storage **808** can include, for example, a magnetic hard disk, a floppy disk, or other persistent read-write data storage technologies, singly or in combination. It can further include mass or archival storage, such as can be provided by CD-ROM or other large-capacity storage technology. (Note that file server **841** provides additional storage capability that processor **805** can use.)

User I/O (input/output) hardware **820** typically includes a visual display monitor such as a CRT or flat-panel display, an alphanumeric keyboard, and a mouse or other pointing device, and optionally can further include a printer, an optical scanner, or other devices for user input and output.

Network I/O hardware **825** provides an interface between computer **800** and the outside world. More specifically, network I/O **825** lets processor **805** communicate via connection **829** with other processors and devices through LAN **840** and through the Internet **880**.

Software components **802** include an operating system **850** and a set of tasks under control of operating system **310**, such as a Java™ application program **860** and, importantly, JVM software **354** and kernel **355**. Operating system **310** also allows processor **805** to control various devices such as persistent storage **808**, user I/O **820**, and network interface **825**. Processor **805** executes the software of operating system **310**, application **860**, JVM **354** and kernel **355** in conjunction with memory **806** and other components of computer system **800**. In an embodiment, software **802** includes network software **302***a*, JVM1, RJVM2 and RJVM3, as illustrated in server **302** of FIG. **3***c*. In an embodiment, Java™ application program **860** is Java™ application **302***c* as illustrated in FIG. **3***c*.

Persons of ordinary skill in the art will appreciate that the system of FIG. **8** is intended to be illustrative, not restrictive, and that a wide variety of computational, communications, and information devices can be used in place of or in addition to what is shown in FIG. **8**. For example, connections through the Internet **880** generally involve packet switching by intermediate router computers (not shown), and computer **800** is likely to access any number of Web servers, including but by no means limited to computer **800** and Web server **881**, during a typical Web client session.

The foregoing description of the preferred embodiments of the present invention has been provided for the purposes of illustration and description. It is not intended to be exhaustive or to limit the invention to the precise forms disclosed. Obviously, many modifications and variations will be apparent to practitioners skilled in the art. The embodiments were chosen and described in order to best explain the principles of the invention and its practical applications, thereby enabling others skilled in the art to understand the invention for various embodiments and with the various modifications as are suited to the particular use contemplated. It is intended that the scope of the invention be defined by the following claims and their equivalents.

What is claimed is:

1. An article of manufacture including an information storage medium wherein is stored information, comprising:

a first set of digital information, including a Java™ virtual machine with a stub having a load balancing software component for selecting a service provider from a plurality of service providers and a failover software component for removing a failed service provider from a list identifying the plurality of service providers, wherein the Java™ virtual machine with the stub is located on a client processing device,

wherein the load balancing software selects a particular service provider, from the list of plurality of service providers, if both an affinity exists for the particular service provider and the particular service provider provides a service requested, and,

wherein an affinity exists for a particular service provider when that particular service provider, or the server associated with the service provider, is currently participating in a transaction between either of the service provider or server and the client processing device.

2. The article of manufacture of claim 1, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the load balancing software component selects a service provider from the list of plurality of service providers in a round robin manner.

3. The article of manufacture of claim 1, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the load balancing software component randomly selects a service provider from the list of service providers.

4. The article of manufacture of claim 1, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the load balancing software component selects a service provider from the list of service providers based upon the load of each service provider.

5. The article of manufacture of claim 1, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the load balancing software component selects a service provider from the list of service providers based upon the data type requested.

6. The article of manufacture of claim 1, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the load balancing software component selects a service provider from the list of service providers based upon the closest physical service provider.

7. The article of manufacture of claim 1, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the load balancing software component selects a service provider from the list of service providers based upon a time period in which each service provider responds.

8. An article of manufacture including an information storage medium wherein is stored information, comprising:

a first set of digital information, including a Java™ virtual machine with a Java™ bean object for selecting a service provider from a plurality of service providers;

wherein the Java™ bean object has a load balancing software component that selects a particular service provider, from the plurality of service providers, if both an affinity exists for the particular service provider and the particular service provider provides a service requested, and,

wherein an affinity exists for a particular service provider when that particular service provider, or the server associated with the service provider, is currently participating in a transaction between either of the service provider or server and the client processing device.

9. The article of manufacture of claim 8, wherein the Java™ bean object has a failover software component for removing a failed service provider from a list of service providers.

10. The article of manufacture of claim 8, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the load balancing software component selects a service provider from the plurality of service providers in a round robin manner.

11. The article of manufacture of claim 8, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the load balancing software component randomly selects a service provider.

12. The article of manufacture of claim 8, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the load balancing software component selects a service provider from the plurality of service providers based upon the load of each service provider.

13. The article of manufacture of claim 8, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the load balancing software component selects a service provider from the plurality of service providers based upon the data type requested.

14. The article of manufacture of claim 8, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the load balancing software component selects a service provider from the plurality of service providers based upon the closest physical service provider.

15. The article of manufacture of claim 8, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the load balancing software component selects a service provider from the plurality of service providers based upon a time period in which each service provider responds.

16. The article of manufacture of claim 8, further comprising:

a second set of digital information, including a stateless session bean.

17. The article of manufacture of claim 8, further comprising:

a second set of digital information, including a stateful session bean.

18. The article of manufacture of claim 8, further comprising:

a second set of digital information, including an entity session bean.

19. An apparatus, comprising:

a processor;

an instruction store, coupled to the processor, comprising an article of manufacture as recited in claim 1; and

a data store, coupled to the processor, wherein an application program can be stored.

20. An apparatus, comprising:

a processor;

an instruction store, coupled to the processor, comprising an article of manufacture as recited in claim 8; and

a data store, coupled to the processor, wherein an application program can be stored.

21. A processing device implemented method, comprising the steps of:

obtaining, by a stub, a list of service providers; and

selecting a service provider for use in a transaction, by the stub, from the list of service providers

wherein the selecting step includes selecting, by the stub, a particular service provider, from the list of plurality of service providers, if both an affinity exists for the particular service provider and the particular service provider provides a service requested, and,

wherein an affinity exists for a particular service provider when that particular service provider, or the server associated with the service provider, is currently participating in the transaction.

22. The method of claim 21, wherein the list of service providers is obtained from a naming service.

23. The method of claim 21, wherein the list of service providers is obtained from a naming service.

24. The method of claim 21, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the step of selecting further includes the step of:

selecting a service provider, by the stub, from the list of service providers in a round robin manner.

25. The method of claim 21, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the step of selecting further includes the step of:

selecting a service provider, by the stub, randomly from the list of service providers.

26. The method of claim 21, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the step of selecting further includes the steps of:

obtaining the load of each service provider, by the stub, in the list of service providers; and,

selecting a service provider, by the stub, based upon the load of each service provider.

27. The method of claim 21, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the step of selecting further includes the steps of:

determining the type of data requested; and,

selecting a service provider, by the stub, from the list of service providers based upon the data type.

28. The method of claim 21, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the step of selecting further includes the steps of:

determining the physical distance to each service provider, by the stub, in the list of service providers; and,

selecting a service provider, by the stub, from the list of service providers based upon on the closest physical distance to the service provider.

29. The method of claim 21, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the step of selecting further includes the steps of:

determining a time period for each service provider, by the stub, in the list of service providers to respond; and,

selecting a service provider from the list of service providers based upon the time period for each service provider to respond.

30. A processing device implemented method, comprising:

obtaining a calling thread;

determining, by a client, if the calling thread has an affinity for a server, wherein an affinity exists for a particular server when that particular server is currently participating in a transaction between the server and the client;

determining if the server provides a service;

obtaining a list of services, wherein the service is in the list of services providers; and,

attempting to obtain the service.

31. The method of claim 30, further comprising:

obtaining a failover method if the service is not available.

32. The method of claim 31, wherein the failover method obtains a next service provider in the list of service providers.

33. The method of claim 31, wherein the failover method obtains a randomly selected service provider in the list of service providers.

34. The method of claim 31, wherein the failover method obtains a service provider with the least load in the list of service providers.

35. The method of claim 31, wherein the failover method obtains a service provider based on a data type in the list of service providers.

36. The method of claim 31, wherein the failover method obtains a service provider based on the closest physical distance to the service provider.

37. The method of claim 31, wherein the failover method obtains a service provider based on a time for a response from a service provider in the plurality of service providers.

38. A method of providing failover in a distributed processing system, to select which service provider within a plurality of service providers should respond to a request from a client to access a service, the method comprising the steps of:

determining whether the client has an affinity for a particular service provider within the plurality of service providers;

if the client does have an affinity for a particular service provider then the substeps of

determining whether the particular service provider can provide the service requested, and,

returning the name of that service provider to the client;

if the client does not have an affinity for a particular service provider, or if the particular service provider is no longer available to provide the service requested, then the substeps of

selecting a new service provider from the plurality of service providers, and,

returning the name of the new service provider to the client; and,

allowing the client to access the service using the named service provider, if the named service provider exists and can provide the service.

39. The method of claim 38 wherein said step (B) of determining whether the client has an affinity for a particular service provider includes determining which service provider is coordinating the current transaction between the client and the server, and identifying that service provider as the particular service provider which the client has an affinity for.

40. The method of claim 38 wherein said step of selecting a new service provider includes selecting a new service provider from a list of service providers.

41. The method of claim 38 wherein said step of selecting a new service provider includes selecting a service provider from a list of service providers which is maintained by a naming service.

42. The method of claim 38 wherein said step of selecting a new service provider includes calling a get.next.provider function to obtain and select the next service provider on the list of service providers.

43. The method of claim 38 further comprising, following said step (D) of allowing the client to request service from the named service provider, the additional steps of:

(E) if the named service provider does not exist or cannot provide the service, then calling a failover method that includes the substeps of

identifying the named service provider as a failed service provider,

selecting a failover service provider from the plurality of service providers, and,

returning the name of the failover service provider to the client.

44. The method of claim 38 further comprising the step of:

removing the failed service provider from a list of available service providers within the distributed processing system.

45. The method of claim 38 wherein the service is a database or file system.

46. The method of claim 38, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the system selects a service provider from the list of plurality of service providers in a round robin manner.

47. The method of claim 38, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the system randomly selects a service provider from the list of service providers.

48. The method of claim 38, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the system selects a service provider from the list of service providers based upon the load of each service provider.

49. The method of claim 38, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the system selects a service provider from the list of service providers based upon the data type requested.

50. The method of claim 38, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the system selects a service provider from the list of service providers based upon the closest physical service provider.

51. The method of claim 38, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the system selects a service provider from the list of service providers based upon a time period in which each service provider responds.

52. A method of using load balancing and/or failover in a distributed processing system to select which a service provider within a plurality of service providers can respond to a transaction request from a client to access a service, the method comprising the steps of:

(A) receiving a transaction request from a client to access a service;

(B) determining whether the client has an affinity for a particular service provider within said plurality of service providers;

(C1) if the client does have an affinity for a particular service provider then the substeps of

determining whether the particular service provider can provide the service requested, and,

returning the name of the particular service provider to the client for use by the client in accessing the service;

(C2) if the client does not have an affinity for a particular service provider, then the substeps of

selecting a new service provider from the plurality of service providers,

determining whether the new service provider can provide the service requested, and,

returning the name of the new service provider to the client for use by the client in accessing the service; and,

(D) allowing the client to request service from the named service provider, if the named service provider is available and can provide access to the service requested.

53. The method of claim 52 wherein said step (B) of determining whether the client has an affinity for a particular service provider includes determining which service provider is coordinating the current transaction between the client and the server, and identifying that service provider as the particular service provider which the client has an affinity for.

54. The method of claim 52 wherein said step of selecting a new service provider includes selecting a new service provider from a list of service providers.

55. The method of claim 52 wherein said step of selecting a new service provider includes selecting a service provider from a list of service providers which is maintained by a naming service.

56. The method of claim 52 wherein said step of selecting a new service provider includes calling a get.next.provider function to obtain and select the next service provider on the list of service providers.

57. The method of claim 52 further comprising, following said step (D) of allowing the client to request service from the named service provider, the additional steps of:

(E) if the named service provider does not exist or cannot provide the service, then calling a failover method that includes the substeps of

identifying the named service provider as a failed service provider,

selecting a failover service provider from the plurality of service providers, and,

returning the name of the failover service provider to the client.

58. The method of claim 52 further comprising the step of:

removing the failed service provider from a list of available service providers within the distributed processing system.

59. The method of claim 52 wherein the service is a database or file system.

60. The method of claim 52, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the system selects a service provider from the list of plurality of service providers in a round robin manner.

61. The method of claim 52, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the system randomly selects a service provider from the list of service providers.

62. The method of claim 52, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the system selects a service provider from

the list of service providers based upon the load of each service provider.

63. The method of claim **52**, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the system selects a service provider from the list of service providers based upon the data type requested.

64. The method of claim **52**, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the system selects a service provider from the list of service providers based upon the closest physical service provider.

65. The method of claim **52**, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the system selects a service provider from the list of service providers based upon a time period in which each service provider responds.

66. A system for load balancing and failover of requests by a client to access a service in a distributed processing system comprising:

a handler for receiving requests from a client to access a service;

a plurality of service providers for providing client access to the service;

software code that performs the method of

determining whether the client has an affinity for a particular service provider within the plurality of service providers;

if the client does have an affinity for a particular service provider then the substeps of

determining whether the particular service provider can provide the service requested, and,

returning the name of that service provider to the client;

if the client does not have an affinity for a service provider, then the substeps of selecting a new service provider from the plurality of service providers, and,

returning the name of the new service provider to the client; and,

allowing the client to access the service using the named service provider, if the named service provider exists and can provide the service.

67. The system of claim **66** further comprising:

a list of currently available service providers, and,

wherein a failed service provider is removed from the list of service providers.

68. The system of claim **66** wherein the service is a database or file system.

69. The system of claim **66**, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the system selects a service provider from the list of plurality of service providers in a round robin manner.

70. The system of claim **66**, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the system randomly selects a service provider from the list of service providers.

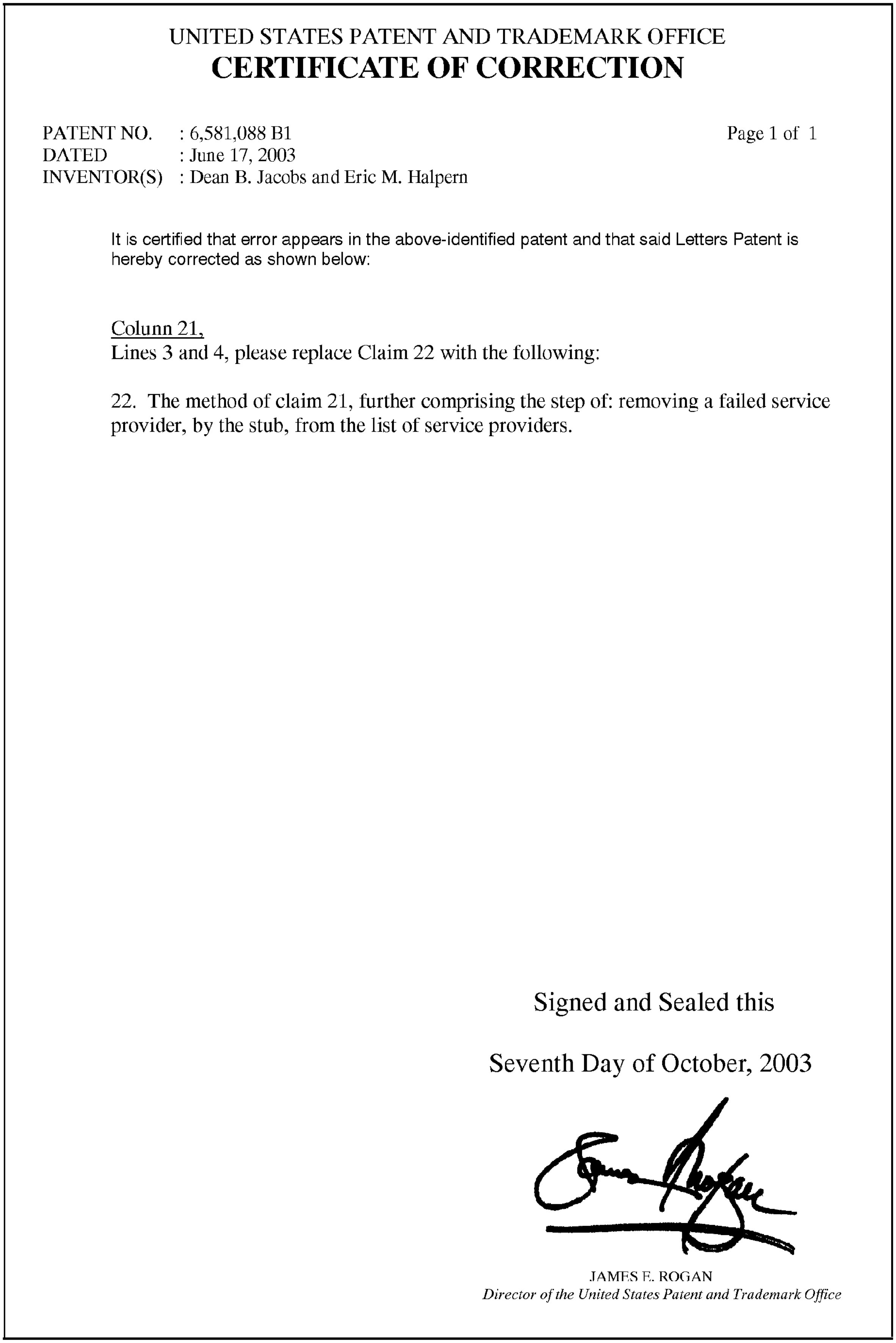
71. The system of claim **66**, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the system selects a service provider from the list of service providers based upon the load of each service provider.

72. The system of claim **66**, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the system selects a service provider from the list of service providers based upon the data type requested.

73. The system of claim **66**, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the system selects a service provider from the list of service providers based upon the closest physical service provider.

74. The system of claim **66**, wherein if no affinity for a service provider exists, or if the particular service provider for which the affinity exists does not provide the service requested, then the system selects a service provider from the list of service providers based upon a time period in which each service provider responds.

* * * * *



UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO.	: 6,581,088 B1	Page 1 of 1
DATED	: June 17, 2003	
INVENTOR(S)	: Dean B. Jacobs and Eric M. Halpern	

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

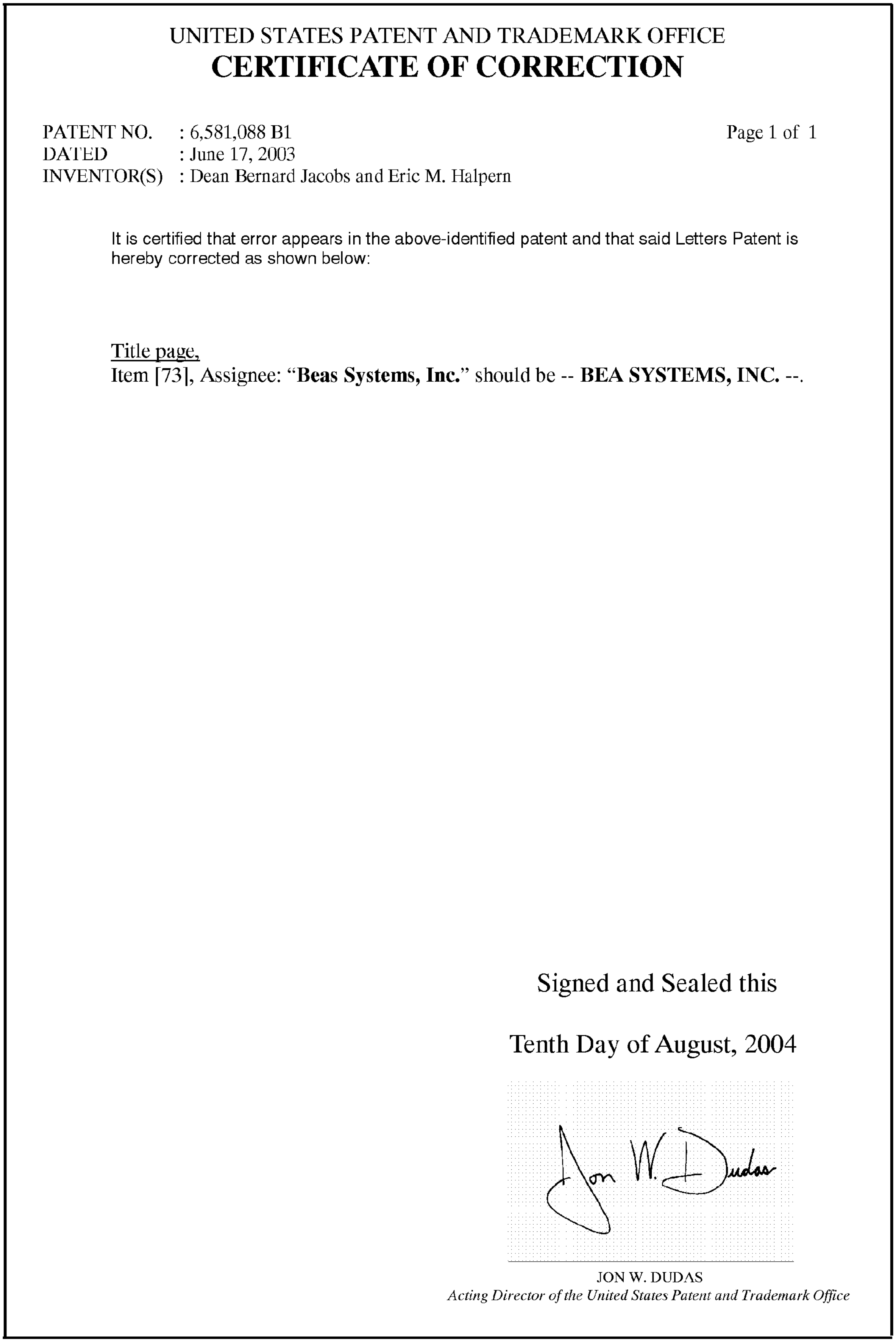
Colunn 21,
Lines 3 and 4, please replace Claim 22 with the following:

22. The method of claim 21, further comprising the step of: removing a failed service provider, by the stub, from the list of service providers.

Signed and Sealed this

Seventh Day of October, 2003

JAMES E. ROGAN
Director of the United States Patent and Trademark Office



UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 6,581,088 B1 Page 1 of 1
DATED : June 17, 2003
INVENTOR(S) : Dean Bernard Jacobs and Eric M. Halpern

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

Title page,
Item [73], Assignee: "**Beas Systems, Inc.**" should be -- **BEA SYSTEMS, INC.** --.

Signed and Sealed this

Tenth Day of August, 2004

Jon W. Dudas

JON W. DUDAS
Acting Director of the United States Patent and Trademark Office

US005999979A

United States Patent [19]

Vellanki et al.

[11] **Patent Number:** **5,999,979**

[45] **Date of Patent:** **Dec. 7, 1999**

[54] **METHOD AND APPARATUS FOR DETERMINING A MOST ADVANTAGEOUS PROTOCOL FOR USE IN A COMPUTER NETWORK**

[75] Inventors: **Srinivas Prasad Vellanki**, Milpitas; **Anthony William Cannon**, Mountain View; **Hemanth Srinivas Ravi**, Milpitas; **Anders Edgar Klemets**, Sunnyvale, all of Calif.

[73] Assignee: **Microsoft Corporation**, Redmond, Wash.

[21] Appl. No.: **08/818,769**

[22] Filed: **Mar. 14, 1997**

Related U.S. Application Data

[60] Provisional application No. 60/036,661, Jan. 30, 1997, and provisional application No. 60/036,662, Jan. 30, 1997.

[51] **Int. Cl.**^{6} **G06F 15/16**

[52] **U.S. Cl.** **709/232**; 709/237

[58] **Field of Search** 395/200.57, 200.58, 395/200.6, 200.76, 200.62; 710/11; 709/228, 227, 230, 231, 232, 237

[56] **References Cited**

U.S. PATENT DOCUMENTS

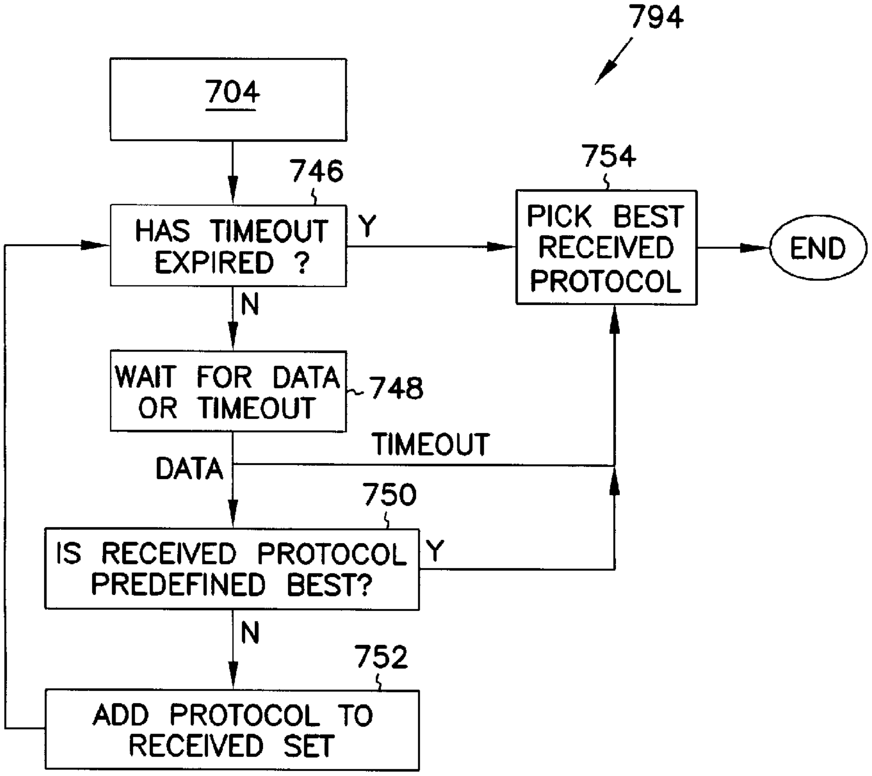
4,931,250	6/1990	Greszczuk	375/8
5,202,899	4/1993	Walsh	375/8
5,349,649	9/1994	Iijima	395/275
5,557,724	9/1996	Sampat et al.	395/157
5,687,174	11/1997	Edem et al.	370/446

Primary Examiner—Stuart S. Levy
Assistant Examiner—Kenneth W. Fields
Attorney, Agent, or Firm—Schwegman, Lundberg, Woessner & Kluth P.A.

[57] **ABSTRACT**

A method in a computer network for automatically detecting a most advantageous protocol for communication by a client computer, said client computer being configured to be coupled to a server computer via a computer network. The method includes initiating a plurality of protocol threads for sending from the client computer to the server computer, a plurality of data requests. Each of the data requests employs a different protocol and a different connection. The data requests are configured to solicit, responsive to the data requests, a set of responses from the server computer. Each of the responses employs a protocol associated with a respective one of the data requests. The method further includes receiving at the client computer at least a subset of the responses. The method also includes initiating a control thread at the client computer. The control thread monitors the subset of the responses as each response is received from the server computer to select the most advantageous protocol from protocols associated with the subset of the responses, wherein the most advantageous protocol is determined based on a predefined protocol priority.

23 Claims, 11 Drawing Sheets


794
704
746
HAS TIMEOUT EXPIRED ?
Y
N
754
PICK BEST RECEIVED PROTOCOL
END
WAIT FOR DATA OR TIMEOUT
748
TIMEOUT
DATA
750
IS RECEIVED PROTOCOL PREDEFINED BEST?
Y
N
752
ADD PROTOCOL TO RECEIVED SET

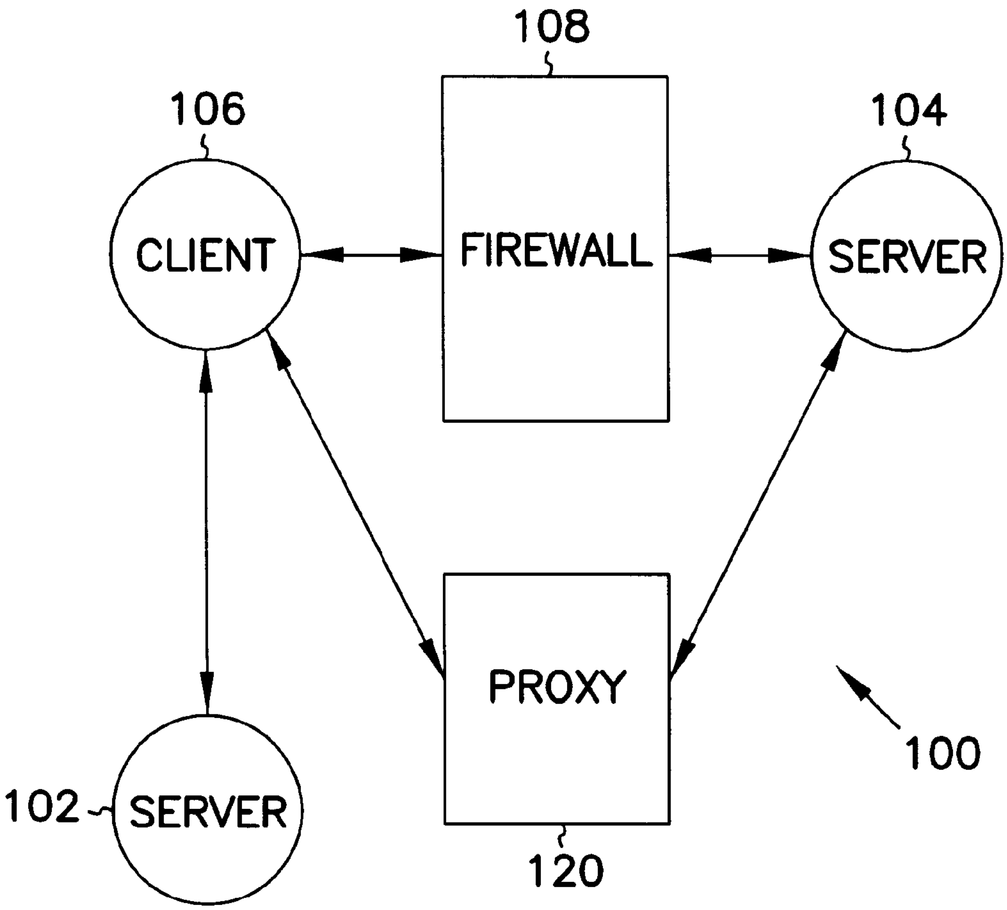


FIG. 1

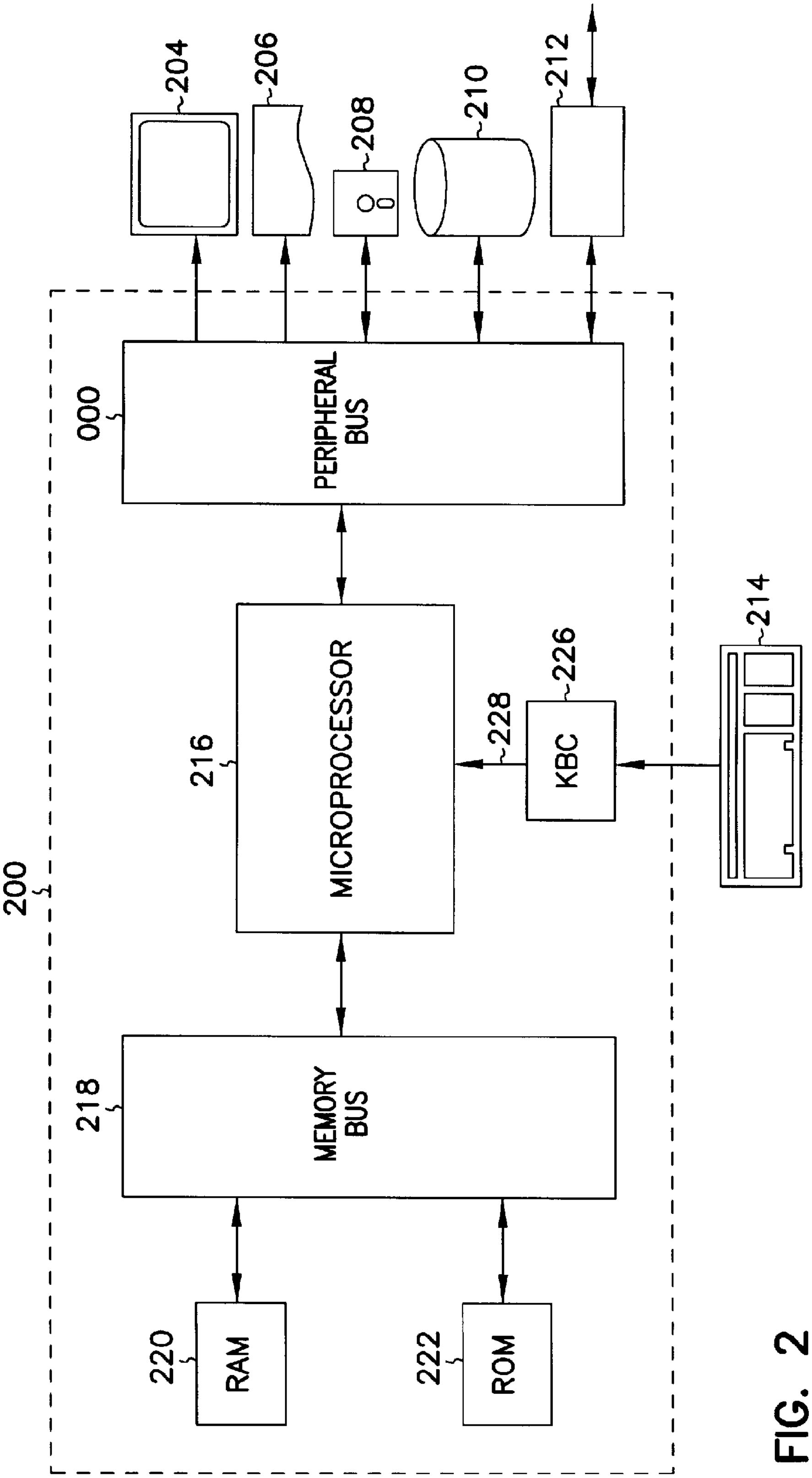

204
206
208
210
212
000
PERIPHERAL BUS
216
MICROPROCESSOR
228
226
KBC
214
200
218
MEMORY BUS
220
RAM
222
ROM

FIG. 2

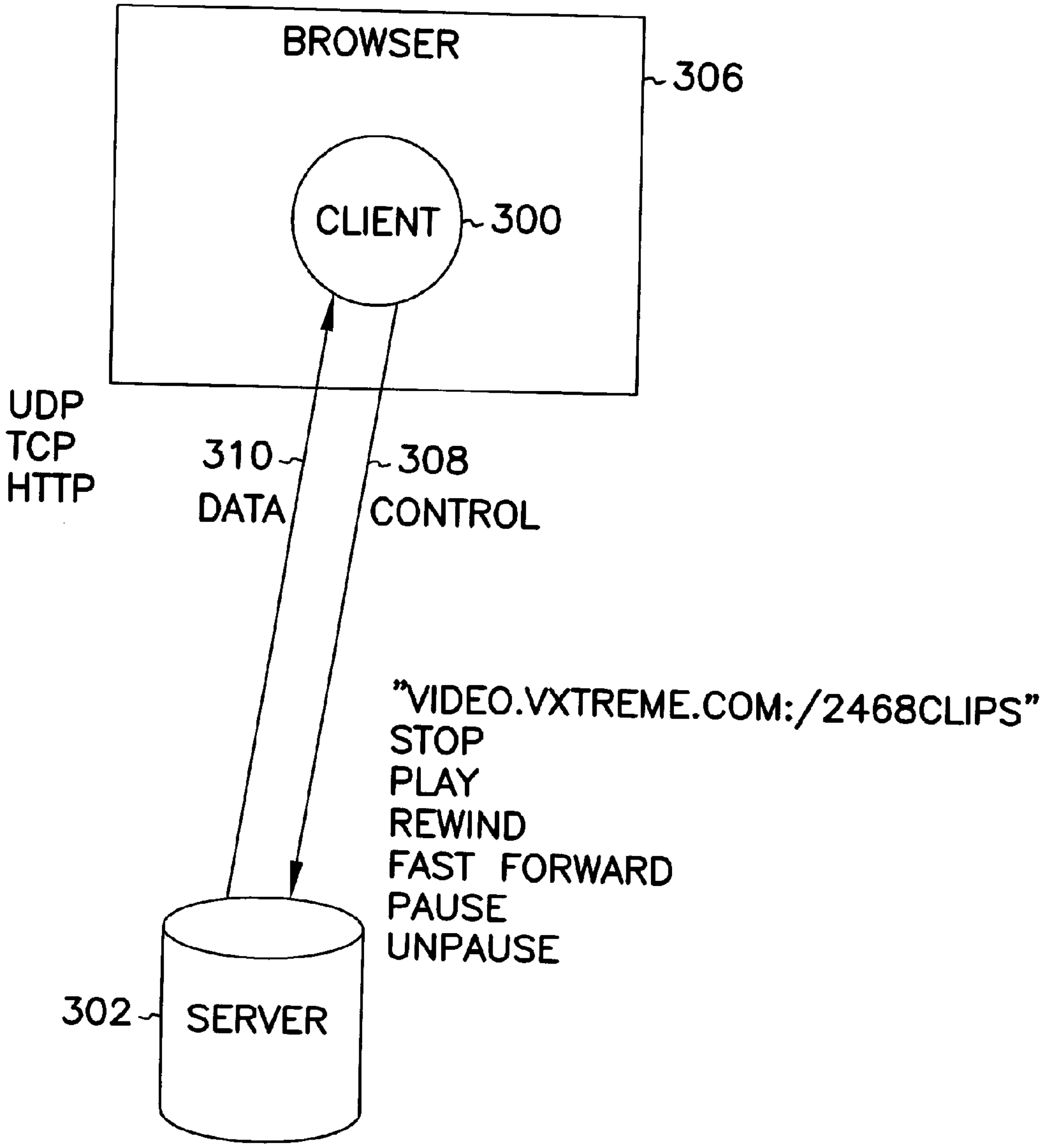

BROWSER
306
CLIENT
300
UDP
TCP
HTTP
310
DATA
308
CONTROL
"VIDEO.VXTREME.COM:/2468CLIPS"
STOP
PLAY
REWIND
FAST FORWARD
PAUSE
UNPAUSE
302
SERVER

FIG. 3

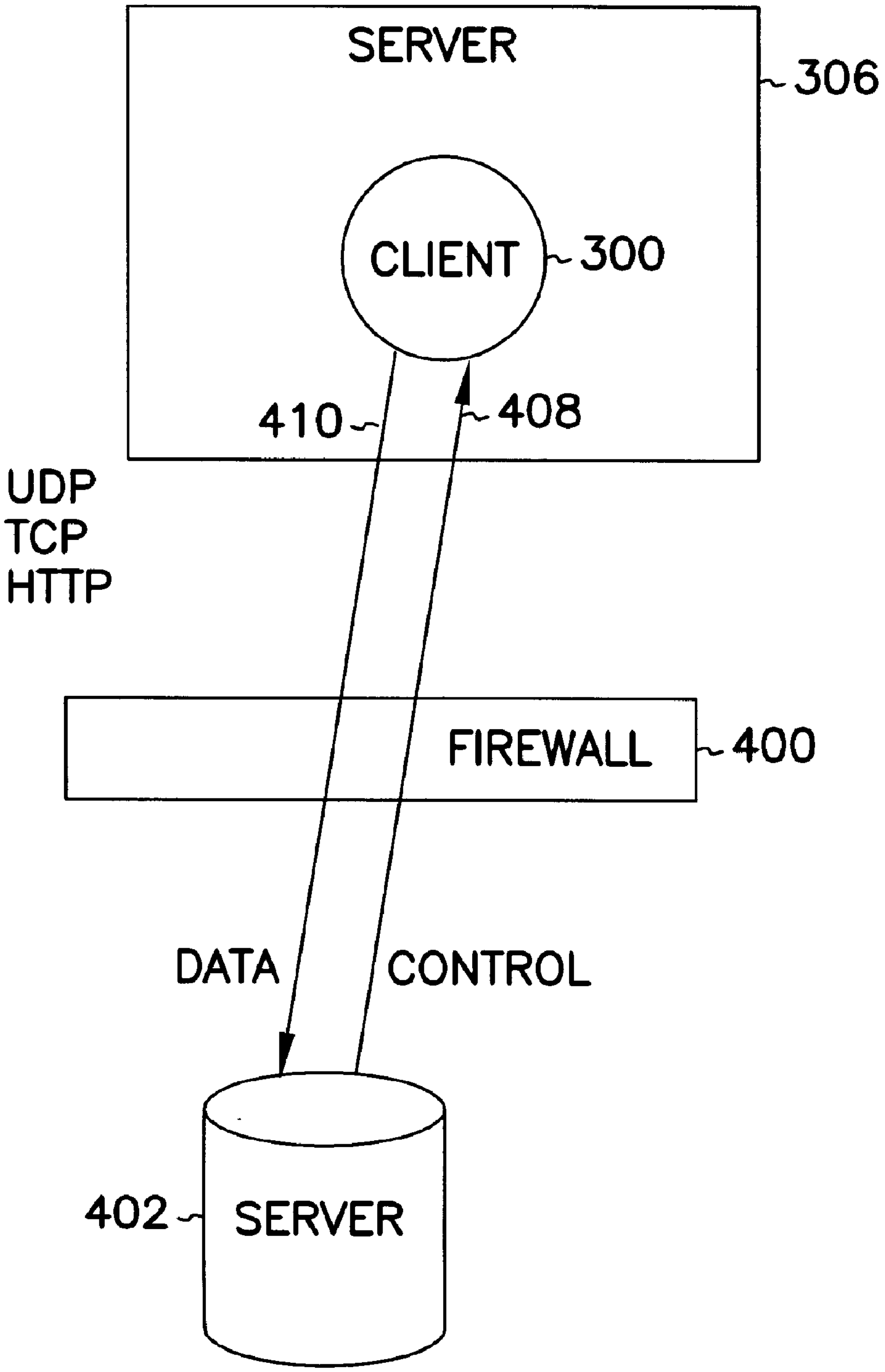

SERVER
306
CLIENT
300
410
408
UDP
TCP
HTTP
FIREWALL
400
DATA
CONTROL
402
SERVER

FIG. 4

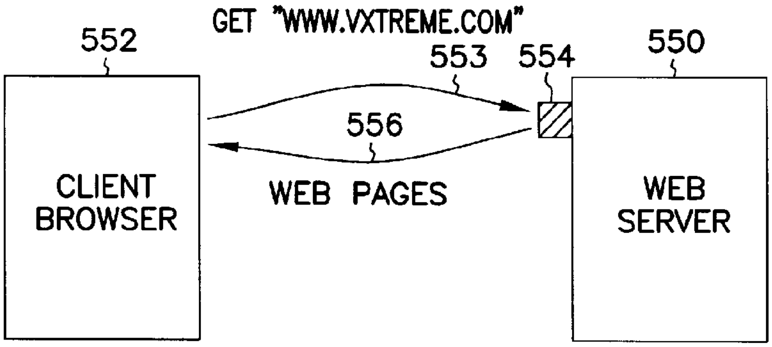


FIG. 5A (PRIOR ART)

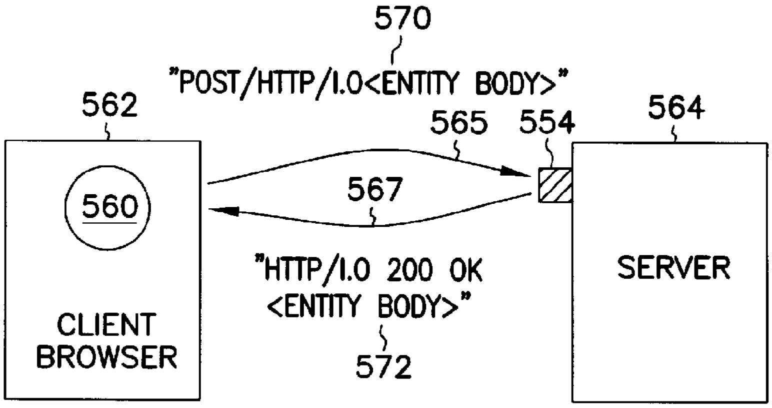


FIG. 5B

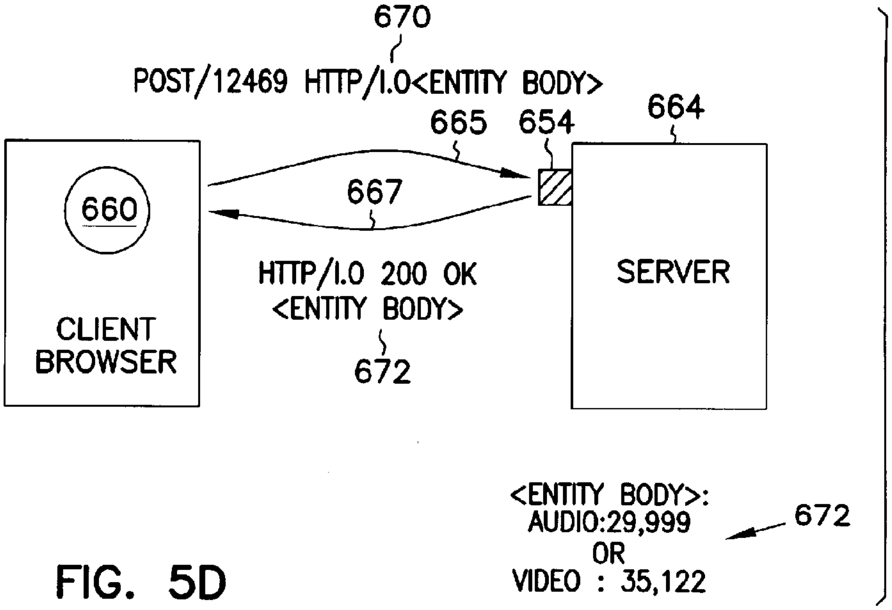


FIG. 5D

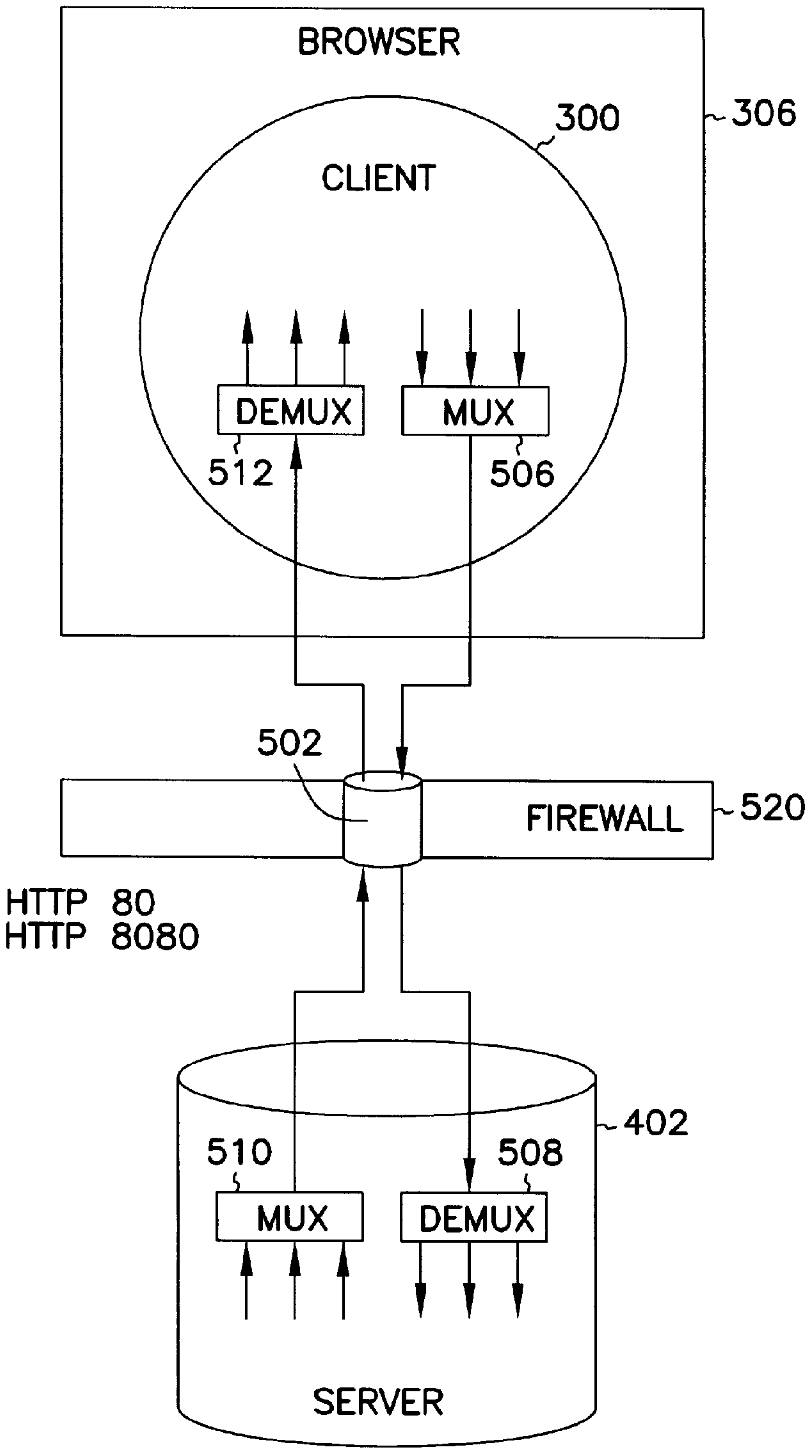

BROWSER
300
306
CLIENT
DEMUX
MUX
512
506
502
FIREWALL
520
HTTP 80
HTTP 8080
402
510
508
MUX
DEMUX
SERVER

FIG. 5C

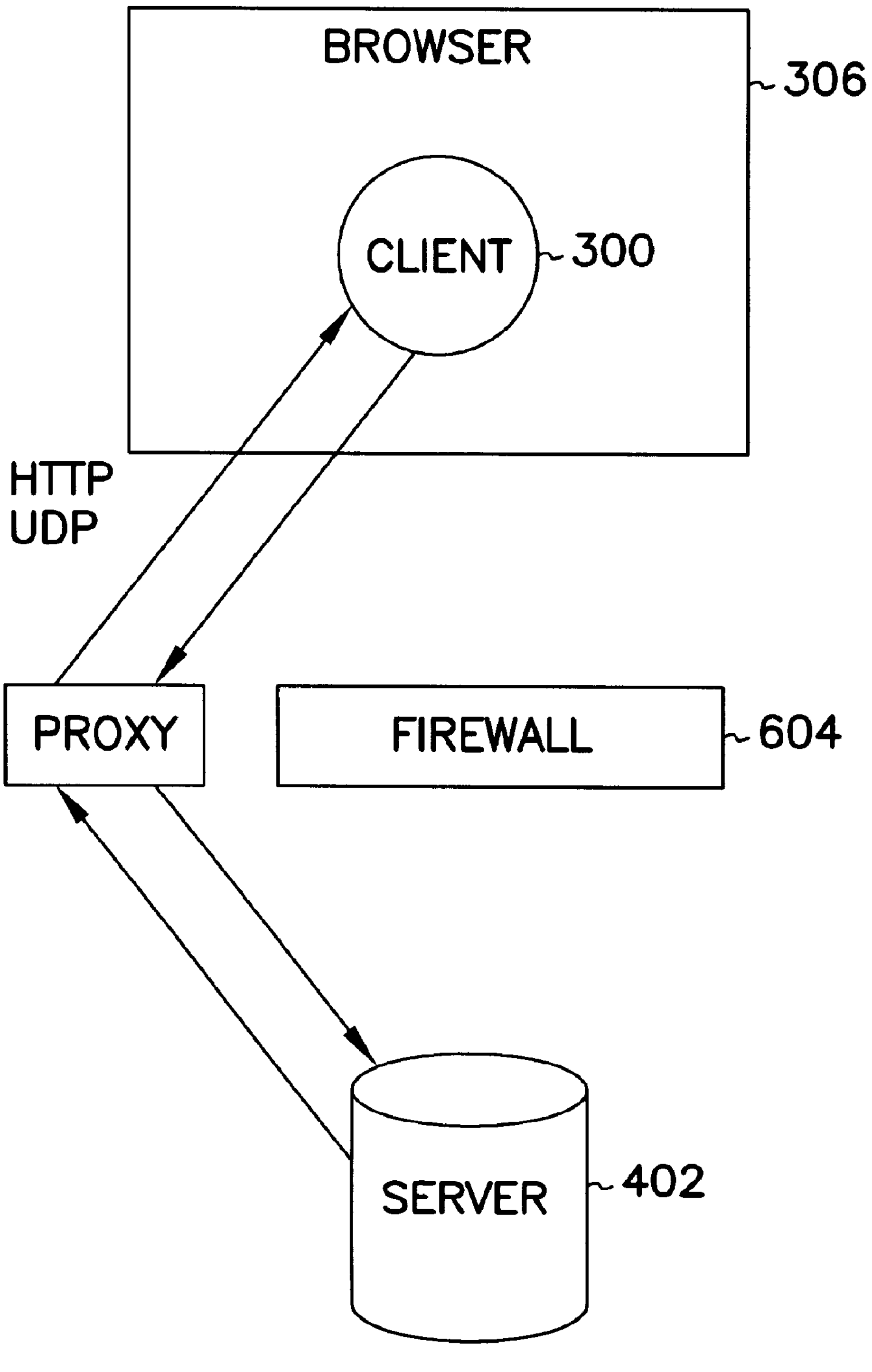

BROWSER
306
CLIENT
300
HTTP
UDP
PROXY
FIREWALL
604
SERVER
402

FIG. 6

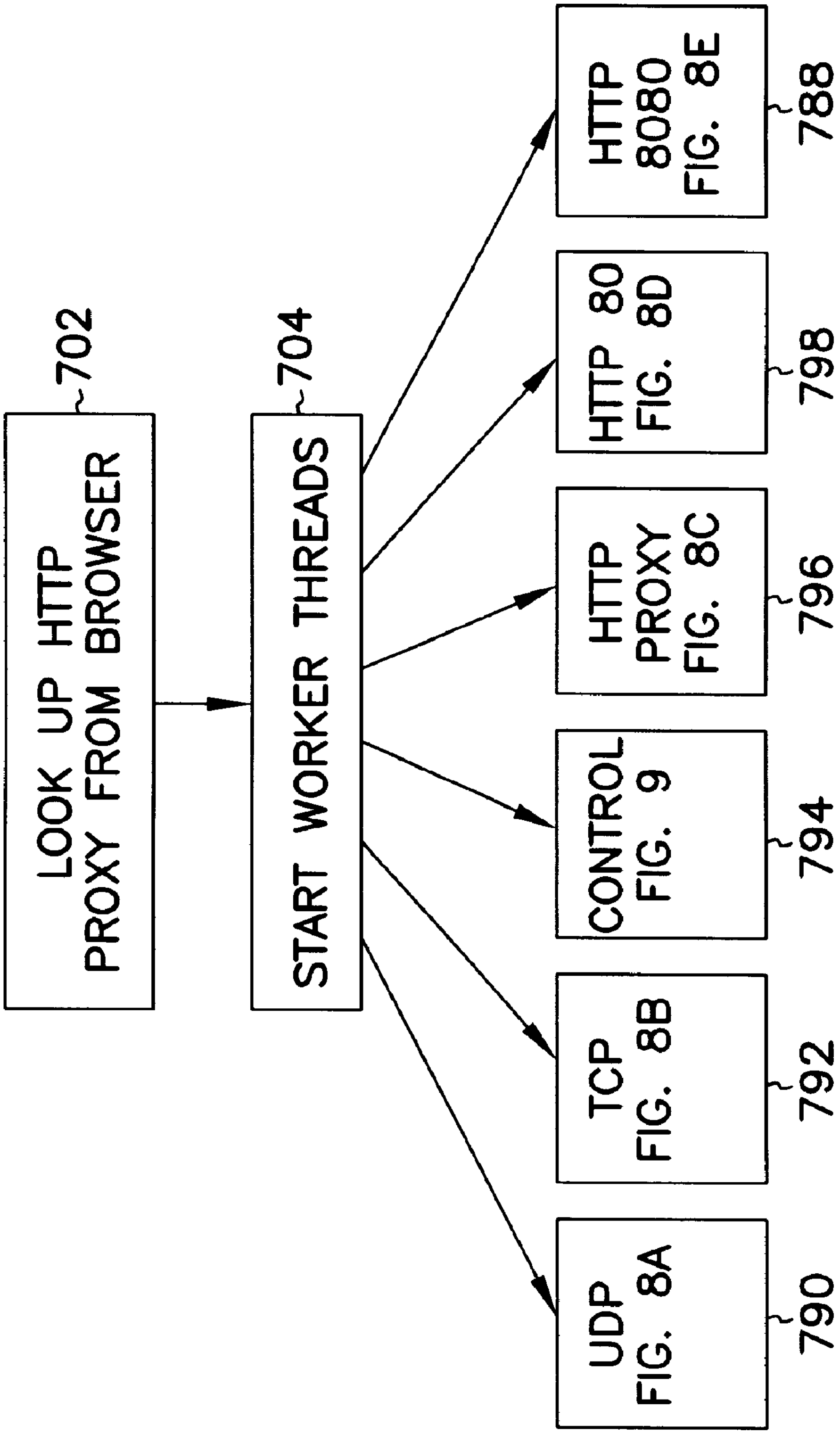

LOOK UP HTTP PROXY FROM BROWSER
702
START WORKER THREADS
704
UDP FIG. 8A
790
TCP FIG. 8B
792
CONTROL FIG. 9
794
HTTP PROXY FIG. 8C
796
HTTP 80 FIG. 8D
798
HTTP 8080 FIG. 8E
788

FIG. 7

FIG. 8A

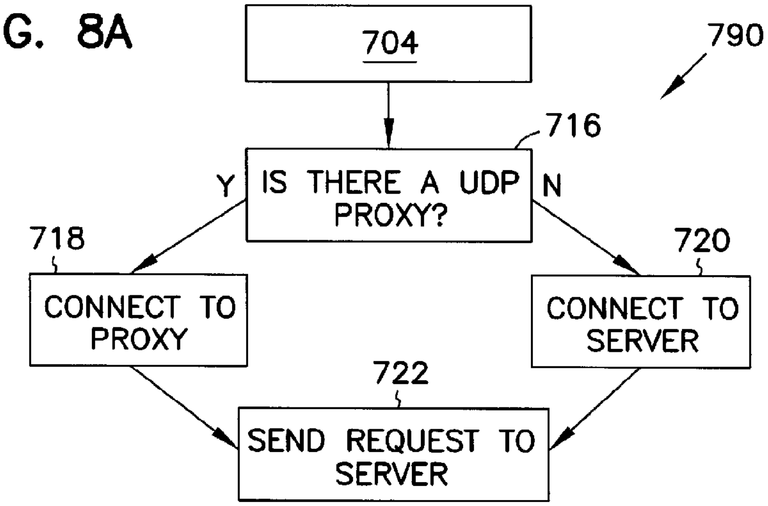


FIG. 8B

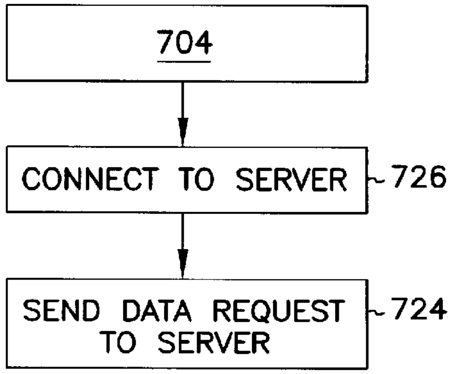


FIG. 8C

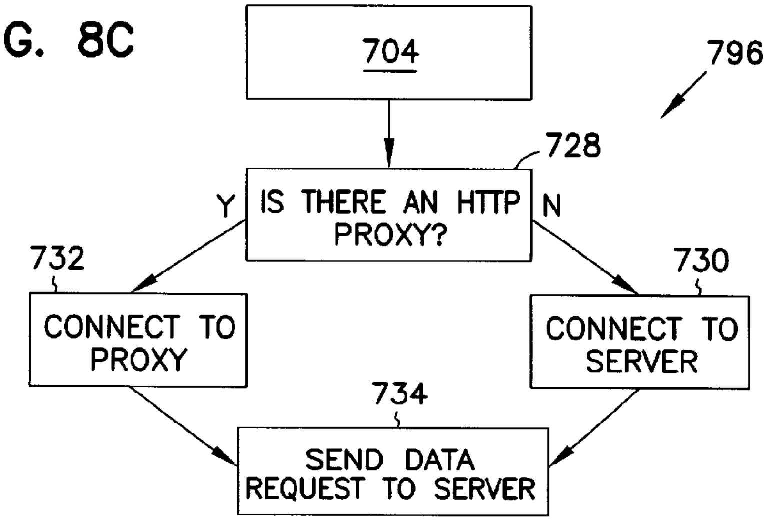


FIG. 8D

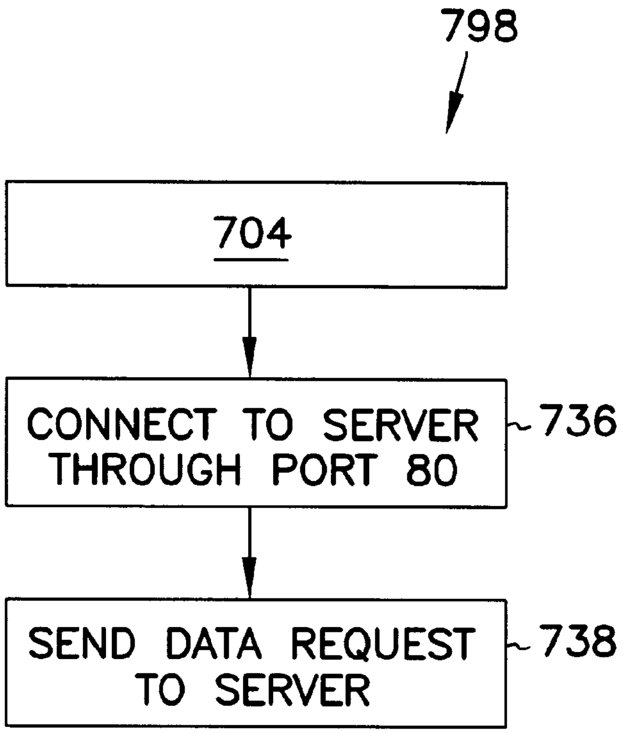
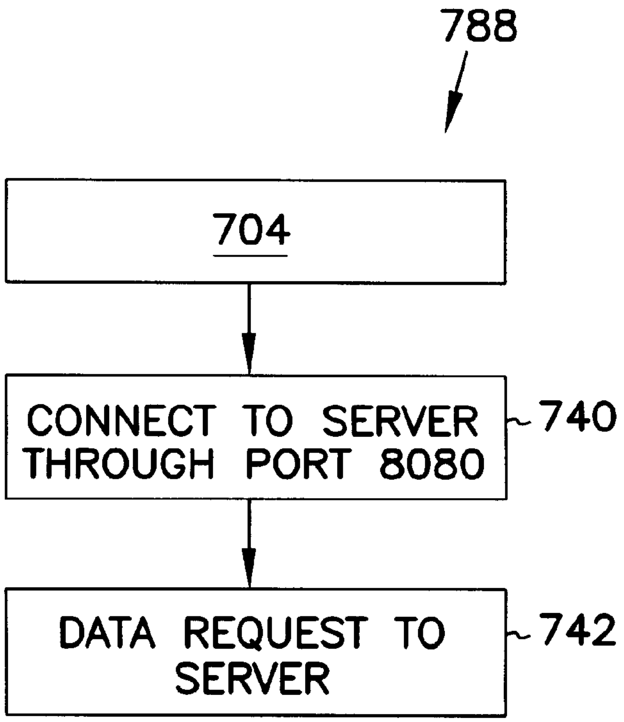

798
704
CONNECT TO SERVER THROUGH PORT 80
736
SEND DATA REQUEST TO SERVER
738

FIG. 8E


788
704
CONNECT TO SERVER THROUGH PORT 8080
740
DATA REQUEST TO SERVER
742

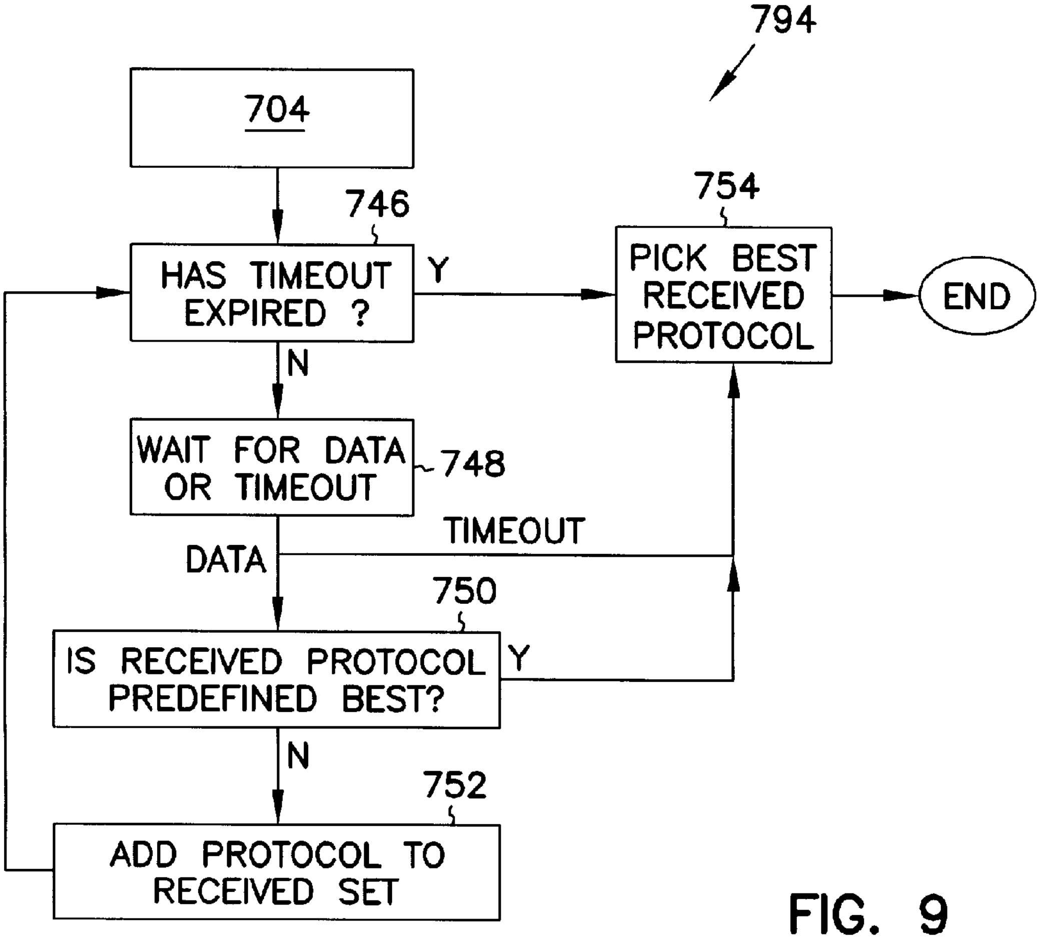

794
704
746
HAS TIMEOUT EXPIRED ?
Y
754
PICK BEST RECEIVED PROTOCOL
END
N
WAIT FOR DATA OR TIMEOUT
748
TIMEOUT
DATA
750
IS RECEIVED PROTOCOL PREDEFINED BEST?
Y
N
752
ADD PROTOCOL TO RECEIVED SET

FIG. 9

METHOD AND APPARATUS FOR DETERMINING A MOST ADVANTAGEOUS PROTOCOL FOR USE IN A COMPUTER NETWORK

This application claims priority under 35 U.S.C 119 (e) of a provisional application entitled "VCR CONTROL FUNCTIONS" filed Jan. 30, 1997 by inventors Anthony W. Cannon, Anders E. Klemets, Hemanth S. Ravi, and David del Val (application Ser. No. 60/036,661) and a provisional application entitled "METHODS AND APPARATUS FOR AUTODETECTING PROTOCOLS IN A COMPUTER NETWORK" filed Jan. 30, 1997 by inventors Anthony W. Cannon, Anders E. Klemets, Hemanth S. Ravi, and David del Val (application Ser. No. 60/036,662).

CROSS REFERENCE TO RELATED APPLICATIONS

This application is related to co-pending U.S. application Ser. No. 08/818,805, filed on Mar. 14, 1997, entitled "Method and Apparatus for Implementing Motion Detection in Video Compression", U.S. application Ser. No. 08/819,507, filed Mar. 14, 1997, entitled "Digital Video Signal Encoder and Encoding Method", U.S. application Ser. No. 08/818,804, filed on Mar. 14, 1997, entitled "Production of a Video Stream with Synchronized Annotations over a Computer Network", U.S. application Ser. No. 08/819,586, filed on Mar. 14, 1997, entitled "Method and Apparatus for Implementing Control Functions in a Streamed Video Display System", U.S. application Ser. No. 08/818,769, filed on Mar. 14, 1997, entitled "Method and Apparatus for Automatically Detecting Protocols in a Computer Network," U.S. application Ser. No.08/818,127, filed on Mar. 14, 1997, entitled "Dynamic Bandwidth Selection for Efficient Transmission of Multimedia Streams in a Computer Network," U.S. application Ser. No. 08/819,585, filed on Mar. 14, 1997, entitled "Streaming and Display of a Video Stream with Synchronized Annotations over a Computer Network", U.S. application Ser. No. 08/818,664, filed on Mar. 14, 1997, entitled "Selective Retransmission for Efficient and Reliable Streaming of Multimedia Packets in a Computer Network", U.S. application Ser. No. 08/819,579, filed Mar. 14, 1997, entitled "Method and Apparatus for Table-Based Compression with Embedded Coding", U.S. application Ser. No. 08/818,826, filed on Mar. 14, 1997, entitled "Digital Video Signal Encoder and Encoding Method", all filed concurrently herewith, U.S. application Ser. No. 08/822,156, filed on Mar. 17, 1997, entitled "Method and Apparatus for Communication Media Commands and Data Using the HTTP Protocol", provisional U.S. application Ser. No. 60/036,662, filed on Jan. 30, 1997, entitled "Methods and Apparatus for Autodetecting Protocols in a Computer Network" U.S. application Ser. No. 08/625,650, filed on Mar. 29, 1996, entitled "Table-Based Low-Level Image Classification System", U.S. application Ser. No. 08/714,447, filed on Sep. 16, 1996, entitled "Multimedia Compression System with Additive Temporal Layers", and is a continuation-in-part of U.S. application Ser. No. 08/623,299, filed on Mar. 28, 1996, entitled "Table-Based Compression with Embedded Coding", which are all incorporated by reference in their entirety for all purposes.

BACKGROUND OF THE INVENTION

The present invention relates to data communication in a computer network. More particularly, the present invention relates to improved methods and apparatus for permitting a client computer in a client-server architecture computer network to automatically detect the most advantageous protocol, among the protocols available, for use in communicating with the server irrespective whether there exist firewalls or proxies in the network.

Client-server architectures are well known to those skilled in the computer art. For example, in a typical computer network, one or more client computers may be coupled to any number of server computers. Client computers typically refer to terminals or personal computers through which end users interact with the network. Server computers typically represent nodes in the computer network where data, application programs, and the like, reside. Server computers may also represent nodes in the network for forwarding data, programs, and the likes from other servers to the requesting client computers.

To facilitate discussion, FIG. **1** illustrates a computer network **100**, representing for example a subset of an international computer network popularly known as the Internet. As is well known, the Internet represents a well-known international computer network that links, among others, various military, governmental, educational, nonprofit, industrial and financial institutions, commercial enterprises, and individuals. There are shown in FIG. **1** a server **102**, a server **104**, and a client computer **106**. Server computer **104** is separated from client computer **106** by a firewall **108**, which may be implemented in either software or hardware, and may reside on a computer and/or circuit between client computer **106** and server computer **104**.

Firewall **108** may be specified, as is well known to those skilled in the art, to prevent certain types of data and/or protocols from traversing through it. The specific data and/or protocols prohibited or permitted to traverse firewall **108** depend on the firewall parameters, which are typically set by a system administrator responsible for the maintenance and security of client computer **106** and/or other computers connected to it, e.g., other computers in a local area network. By way of example, firewall **108** may be set up to prevent TCP, UDP, or HTTP (Transmission Control Protocol, User Datagram Protocol, and Hypertext Transfer Protocol, respectively) data and/or other protocols from being transmitted between client computer **106** and server **104**. The firewalls could be configured to allow specific TCP or UDP sessions, for example outgoing TCP connection to certain ports, UDP sessions to certain ports, and the like.

Without a firewall, any type of data and/or protocol may be communicated between a client computer and a server computer if appropriate software and/or hardware are employed. For example, server **102** resides on the same side of firewall **108** as client computer **106**, i.e., firewall **108** is not disposed in between the communication path between server **102** and client computer **106**. Accordingly, few, if any, of the protocols that client computer **106** may employ to communicate with server **102** may be blocked.

As is well known to those skilled in the art, some computer networks may be provided with proxies, i.e., software codes or hardware circuitries that facilitate the indirect communication between a client computer and a server around a firewall. With reference to FIG. **1**, for example, client computer **106** may communicate with server **104** through proxy **120**. Through proxy **120**, HTTP data, which may otherwise be blocked by firewall **108** for the purpose of this example, may be transmitted between client computer **106** and server computer **104**.

In some computer networks, one or more protocols may be available for communication between the client computer and the server computer. For certain applications, one of these protocols, however, is often more advantageous, i.e., suitable, than others. By way of example, in applications involving real-time data rendering (such as rendering audio, video, and/or annotation data as they are streamed from a server, as described in above-referenced U.S. patent application Ser. Nos. 08/818,804 and 08/819,585 (Atty: Docket No.: VXT710)), it is highly preferable that the client computer executing that application selects a protocol that permits the greatest degree of control over the transmission of data packets and/or enables data transmission to occur at the highest possible rate. This is because these applications are fairly demanding in terms of their bit rate and connection reliability requirements. Accordingly, the quality of the data rendered, e.g., the video and/or audio clips played, often depends on whether the user has successfully configured the client computer to receive data from the server computer using the most advantageous protocol available.

In the prior art, the selection of the most advantageous protocol for communication between client computer **106** and server computer **104** typically requires a high degree of technical sophistication on the part of the user of client computer **106**. By way of example, it is typically necessary in the prior art for the user of client computer **106** to understand the topology of computer network **100**, the protocols available for use with the network, and/or the protocols that can traverse firewall **108** before that user can be expected to configure his client computer **106** for communication.

This level of technical sophistication is, however, likely to be beyond that typically possessed by an average user of client computer **106**. Accordingly, users in the prior art often find it difficult to configure their client computers even for simple communication tasks with the network. The difficulties may be encountered for example during the initial setup or whenever there are changes in the topology of computer network **100** and/or in the technology employed to transmit data between client computer **106** and server **104**. Typically, expert and expensive assistance is required, if such assistance is available at all in the geographic area of the user.

Furthermore, even if the user can configure client computer **106** to communicate with server **104** through firewall **108** and/or proxy **120**, there is no assurance that the user of client computer **106** has properly selected, among the protocols available, the most advantageous protocol communication (e.g., in terms of data transmission rate, transmission control, and the like). As mentioned earlier, the ability to employ the most advantageous protocol for communication, while desirable for most networking applications, and is particularly critical in applications such as real-time data rendering (e.g., rendering of audio, video, and/or annotation data as they are receive from the server). If a less than optimal protocol is chosen for communication, the quality of the rendered data ,e.g., the video clips and/or audio clips, may suffer.

In view of the foregoing, there are desired improved techniques for permitting a client computer in a client-server network to efficiently, automatically, and appropriately select the most advantageous protocol to communicate with a server computer.

SUMMARY OF THE INVENTION

The invention relates, in one embodiment, to a method in a computer network for automatically detecting a most advantageous protocol for communication by a client computer. The client computer is configured to be coupled to a server computer via a computer network. The method includes sending from the client computer to the server computer, a plurality of data requests. Each of the data requests employs a different protocol and a different connection. The data requests are configured to solicit, responsive to the data requests, a set of responses from the server computer. Each of the responses employs a protocol associated with a respective one of the data requests.

The method further includes receiving at the client computer at least a subset of the responses. The method also includes monitoring the subset of the responses as each response is received from the server computer to select the most advantageous protocol from protocols associated with the subset of the responses, wherein the most advantageous protocol is determined based on a predefined protocol priority.

In another embodiment, the invention relates to a method in a computer network for automatically detecting a most advantageous protocol for communication by a client computer. The client computer is configured to be coupled to a server computer via a computer network. The method includes sending from the client computer to the server computer, a plurality of data requests. Each of the data requests employs a different protocol and a different connection.

The method further includes receiving at least a subset of the data requests at the server computer. The method additionally includes sending a set of responses from the server computer to the client computer. The set of responses is responsive to the subset of the data requests. Each of the responses employs a protocol associated with a respective one of the subset of the data requests. The method also includes receiving at the client computer at least a subset of the responses. There is further included selecting, for the communication between the client computer and the server computer, the most advantageous protocol from protocols associated with the subset of the responses, wherein the most advantageous protocol is determined based on a predefined protocol priority.

In yet another embodiment, the invention relates to a computer readable medium containing computer-readable instructions for automatically detecting a most advantageous protocol for communication by a client computer. The client computer is configured to be coupled to a server computer via a computer network. The computer-readable instructions comprise computer readable instructions for sending in a substantially parallel manner, from the client computer to the server computer, a plurality of data requests. Each of the data requests employs a different protocol and a different connection. The data requests are configured to solicit, responsive to the data requests, a set of responses from the server computer. Each of the responses employs a protocol associated with a respective one of the data requests.

The computer readable medium further includes computer readable instructions for receiving at the client computer at least a subset of the responses. There is further included computer readable instructions for monitoring the subset of the responses as each response is received from the server computer to select the most advantageous protocol from protocols associated with the subset of the responses, wherein the most advantageous protocol is determined based on a predefined protocol priority.

These and other features of the present invention will be described in more detail below in the detailed description of the invention and in conjunction with the following figures.

BRIEF DESCRIPTION OF THE DRAWINGS

To facilitate discussion, FIG. **1** illustrates a computer network, representing for example a portion of an international computer network popularly known as the Internet.

FIG. 2 is a block diagram of an exemplar computer system for carrying out the autodetect technique according to one embodiment of the invention.

FIG. 3 illustrates, in accordance with one embodiment, the control and data connections between a client application and a server computer when no firewall is provided in the network.

FIG. 4 illustrates another network arrangement wherein control and data connections are established through a firewall.

FIGS. 5A–B illustrates another network arrangement wherein media control commands and media data may be communicated between a client computer and a server computer using the HTTP protocol.

FIGS. 5C–D illustrate another network arrangement wherein multiple HTTP control and data connections are multiplexed through a single HTTP port.

FIG. 6 illustrates another network arrangement wherein control and data connections are transmitted between the client application and the server computer via a proxy.

FIG. 7 depicts, in accordance with one embodiment of the present invention, a simplified flowchart illustrating the steps of the inventive autodetect technique.

FIG. 8A depicts, in accordance with one aspect of the present invention, the steps involved in executing the UDP protocol thread of FIG. 7.

FIG. 8B depicts, in accordance with one aspect of the present invention, the steps involved in executing the TCP protocol thread of FIG. 7.

FIG. 8C depicts, in accordance with one aspect of the present invention, the steps involved in executing the HTTP protocol thread of FIG. 7.

FIG. 8D depicts, in accordance with one aspect of the present invention, the steps involved in executing the HTTP 80 protocol thread of FIG. 7.

FIG. 8E depicts, in accordance with one aspect of the present invention, the steps involved in executing the HTTP 8080 protocol thread of FIG. 7.

FIG. 9 illustrates, in accordance with one embodiment of the present invention, the steps involved in executing the control thread of FIG. 7.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

The present invention will now be described in detail with reference to a few preferred embodiments thereof as illustrated in the accompanying drawings. In the following description, numerous specific details are set forth in order to provide a thorough understanding of the present invention. It will be apparent, however, to one skilled in the art, that the present invention may be practiced without some or all of these specific details. In other instances, well known process steps have not been described in detail in order to not unnecessarily obscure the present invention.

In accordance with one aspect of the present invention, the client computer in a heterogeneous client-server computer network (e.g., client computer **106** in FIG. **1**) is provided with an autodetect mechanism. When executed, the autodetect mechanism advantageously permits client computer **106** to select, in an efficient and automatic manner, the most advantageous protocol for communication between the client computer and its server. Once the most advantageous protocol is selected, parameters pertaining to the selected protocol are saved to enable the client computer, in future sessions, to employ the same selected protocol for communication.

In accordance with one particular advantageous embodiment, the inventive autodetect mechanism simultaneously employs multiple threads, through multiple connections, to initiate communication with the server computer, e.g., server **104**. Each thread preferably employs a different protocol and requests the server computer to respond to the client computer using the protocol associated with that thread. For example, client computer **106** may, employing the autodetect mechanism, initiate five different threads, using respectively the TCP, UDP, one of HTTP and HTTP proxy, HTTP through port (multiplex) **80**, and HTTP through port (multiplex) **8080** protocols to request server **104** to respond.

Upon receiving a request, server **104** responds with data using the same protocol as that associated with the thread on which the request arrives. If one or more protocols is blocked and fails to reach server **104** (e.g., by a firewall), no response employing the blocked protocol would of course be transmitted from server **104** to client computer **106**. Further, some of the protocols transmitted from server **104** to client computer **106** may be blocked as well. Accordingly, client computer may receive only a subset of the responses sent from server **104**.

In one embodiment, client computer **106** monitors the set of received responses. If the predefined “best” protocol is received, that protocol is then selected for communication by client computer **106**. The predefined “best” protocol may be defined in advance by the user and/or the application program. If the predefined “best” protocol is, however, blocked (as the request is transmitted from the client computer or as the response is transmitted from the server, for example) the most advantageous protocol may simply be selected from the set of protocols received back by the client computer. In one embodiment, the selection may be made among the set of protocols received back by the client computer within a predefined time period after the requests are sent out in parallel.

The selection of the most advantageous protocol for communication among the protocols received by client computer **106** may be performed in accordance with some predefined priority. For example, in the real-time data rendering application, the UDP protocol may be preferred over TCP protocol, which may be in turned preferred over the HTTP protocol. This is because UDP protocol typically can handle a greater data transmission rate and may allow client computer **106** to exercise a greater degree of control over the transmission of data packets.

HTTP data, while popular nowadays for use in transmitting web pages, typically involves a higher number of overhead bits, making it less efficient relative to the UDP protocol for transmitting real-time data. As is known, the HTTP protocol is typically built on top of TCP. The underlaying TCP protocol typically handles the transmission and retransmission requests of individual data packets automatically. Accordingly, the HTTP protocol tends to reduce the degree of control client computer **106** has over the transmission of the data packets between server **104** and client computer **106**. Of course other priority schemes may exist for different applications, or even for different real-time data rendering applications.

In one embodiment, as client computer **106** is installed and initiated for communication with server **104** for the first time, the autodetect mechanism is invoked to allow client computer **106** to send transmission requests in parallel (e.g., using different protocols over different connections) in the manner discussed earlier. After server **104** responds with data via multiple connections/protocols and the most advantageous protocol has been selected by client computer **106** for communication (in accordance with some predefined priority), the parameters associated with the selected protocol are then saved for future communication.

Once the most advantageous protocol is selected, the autodetect mechanism may be disabled, and future communication between client computer **106** and server **104** may proceed using the selected most advantageous protocol without further invocation of the autodetect mechanism. If the topology of computer network **100** changes and communication using the previously selected "most advantageous" protocol is no longer appropriate, the autodetect mechanism may be executed again to allow client computer **106** to ascertain a new "most advantageous" protocol for communication with server **104**. In one embodiment, the user of client computer **106** may, if desired, initiate the autodetect mechanism at anytime in order to enable client computer **106** to update the "most advantageous" protocol for communication with server **104** (e.g., when the user of client computer **106** has reasons to suspect that the previously selected "most advantageous" protocol is no longer the most optimal protocol for communication).

The inventive autodetect mechanism may be implemented either in software or hardware, e.g., via an IC chip. If implemented in software, it may be carried out by any number of computers capable of functioning as a client computer in a computer network. FIG. 2 is a block diagram of an exemplar computer system **200** for carrying out the autodetect technique according to one embodiment of the invention. Computer system **200**, or an analogous one, may be employed to implement either a client or a server of a computer network. The computer system **200** includes a digital computer **202**, a display screen (or monitor) **204**, a printer **206**, a floppy disk drive **208**, a hard disk drive **210**, a network interface **212**, and a keyboard **214**. The digital computer **202** includes a microprocessor **216**, a memory bus **218**, random access memory (RAM) **220**, read only memory (ROM) **222**, a peripheral bus **224**, and a keyboard controller **226**. The digital computer **200** can be a personal computer (such as an Apple computer, e.g., an Apple Macintosh, an IBM personal computer, or one of the compatibles thereof), a workstation computer (such as a Sun Microsystems or Hewlett-Packard workstation), or some other type of computer.

The microprocessor **216** is a general purpose digital processor which controls the operation of the computer system **200**. The microprocessor **216** can be a single-chip processor or can be implemented with multiple components. Using instructions retrieved from memory, the microprocessor **216** controls the reception and manipulation of input data and the output and display of data on output devices.

The memory bus **218** is used by the microprocessor **216** to access the RAM **220** and the ROM **222**. The RAM **220** is used by the microprocessor **216** as a general storage area and as scratch-pad memory, and can also be used to store input data and processed data. The ROM **222** can be used to store instructions or program code followed by the microprocessor **216** as well as other data.

The peripheral bus **224** is used to access the input, output, and storage devices used by the digital computer **202**. In the described embodiment, these devices include the display screen **204**, the printer device **206**, the floppy disk drive **208**, the hard disk drive **210**, and the network interface **212**, which is employed to connect computer **200** to the network. The keyboard controller **226** is used to receive input from keyboard **214** and send decoded symbols for each pressed key to microprocessor **216** over bus **228**.

The display screen **204** is an output device that displays images of data provided by the microprocessor **216** via the peripheral bus **224** or provided by other components in the computer system **200**. The printer device **206** when operating as a printer provides an image on a sheet of paper or a similar surface. Other output devices such as a plotter, typesetter, etc. can be used in place of, or in addition to, the printer device **206**.

The floppy disk drive **208** and the hard disk drive **210** can be used to store various types of data. The floppy disk drive **208** facilitates transporting such data to other computer systems, and hard disk drive **210** permits fast access to large amounts of stored data.

The microprocessor **216** together with an operating system operate to execute computer code and produce and use data. The computer code and data may reside on the RAM **220**, the ROM **222**, the hard disk drive **220**, or even on another computer on the network. The computer code and data could also reside on a removable program medium and loaded or installed onto the computer system **200** when needed. Removable program mediums include, for example, CD-ROM, PC-CARD, floppy disk and magnetic tape.

The network interface circuit **212** is used to send and receive data over a network connected to other computer systems. An interface card or similar device and appropriate software implemented by the microprocessor **216** can be used to connect the computer system **200** to an existing network and transfer data according to standard protocols.

The keyboard **214** is used by a user to input commands and other instructions to the computer system **200**. Other types of user input devices can also be used in conjunction with the present invention. For example, pointing devices such as a computer mouse, a track ball, a stylus, or a tablet can be used to manipulate a pointer on a screen of a general-purpose computer.

The invention can also be embodied as computer readable code on a computer readable medium. The computer readable medium is any data storage device that can store data which can be thereafter be read by a computer system. Examples of the computer readable medium include read-only memory, random-access memory, CD-ROMs, magnetic tape, optical data storage devices. The computer readable code can also be distributed over a network coupled computer systems so that the computer readable code is stored and executed in a distributed fashion.

FIGS. **3**–**6** below illustrate, to facilitate discussion, some possible arrangements for the transmission and receipt of data in a computer network. The arrangements differ depend on which protocol is employed and the configuration of the network itself. FIG. **3** illustrates, in accordance with one embodiment, the control and data connections between a client application **300** and server **302** when no firewall is provided in the network.

Client application **300** may represent, for example, the executable codes for executing a real-time data rendering program such as the Web Theater Client 2.0, available from VXtreme, Inc. of Sunnyvale, Calif. In the example of FIG. **3**, client application **300** includes the inventive autodetect mechanism and may represent a plug-in software module that may be installed onto a browser **306**. Browser **306** may represent, for example, the application program which the user of the client computer employs to navigate the network. By way of example, browser **306** may represent one of the popular Internet browser programs, such as Netscape™ by

Netscape Communications Inc. of Mountain View, Calif. or Microsoft Explorer by Microsoft Corporation of Redmond, Wash.

When the autodetect mechanism of client application **300** is executed in browser **306** (e.g., during the set up of client application **300**), client application **300** sends a control request over control connection **308** to server **302**. Although multiple control requests are typically sent in parallel over multiple control connections using different protocols as discussed earlier, only one control request is depicted in FIG. **3** to facilitate ease of illustration.

The protocol employed to send the control request over control connection **308** may represent, for example, TCP, or HTTP. If UDP protocol is requested from the server, the request from the client may be sent via the control connection using for example the TCP protocol. Initially, each control request from client application **300** may include, for example, the server name that identifies server **302**, the port through which control connection may be established, and the name of the video stream requested by client application **300**. Server **302** then responds with data via data connection **310**.

In FIG. **3**, it is assumed that no proxies and/or firewalls exist. Accordingly, server **302** responds using the same protocol as that employed in the request. If the request employs TCP, however, server **302** may attempt to respond using either UDP or TCP data connections (depending on the specifics of the request). The response is sent to client application via data connection **310**. If the protocol received by the client application is subsequently selected to be the "most advantageous" protocol, subsequent communication between client application **300** and server **302** may take place via control connection **308** and data connection **310**. Subsequent control requests sent by client application **300** via control connection **308** may include, for example, stop, play, fast forward, rewind, pause, unpause, and the like. These control requests may be utilized by server **302** to control the delivery of the data stream from server **302** to client application **300** via data connection **310**.

It should be noted that although only one control connection and one data connection is shown in FIG. **3** to simplify illustration, multiple control and data connections utilizing the same protocol may exist during a data rendering session. Multiple control and data connections may be required to handle the multiple data streams (e.g., audio, video, annotation) that may be needed in a particular data rendering session. If desired, multiple clients applications **300** may be installed within browser **306**, e.g., to simultaneously render multiple video clips, each with its own sound and annotations.

FIG. **4** illustrates another network arrangement wherein control and data connections are established through a firewall. As mentioned earlier, a firewall may have policies that restrict or prohibit the traversal of certain types of data and/or protocols. In FIG. **4**, a firewall **400** is disposed between client application **300** and server **402**. Upon execution, client application **300** sends control request using a given protocol via firewall **400** to server **402**. Server **402** then responds with data via data connection **410**, again via firewall **400**.

If the data and/or protocol can be received by the client computer through firewall **400**, client application **300** may then receive data from server **402** (through data connection **408**) in the same protocol used in the request. As before, if the request employs the TCP protocol, the server may respond with data connections for either TCP or UDP protocol (depending on the specifics of the request). Protocols that may traverse a firewall may include one or more of the following: UDP, TCP, and HTTP.

In accordance with one aspect of the present invention, the HTTP protocol may be employed to send/receive media data (video, audio, annotation, or the like) between the client and the server. FIG. 5A is a prior art drawing illustrating how a client browser may communicate with a web server using a port designated for communication. In FIG. **5**, there is shown a web server **550**, representing the software module for serving web pages to a browser application **552**. Web server **550** may be any of the commercially available web servers that are available from, for example, Netscape Communications Inc. of Mountain View, Calif. or Microsoft Corporation of Redmond, Wash. Browser application **552** represents for example the Netscape browser from the aforementioned Netscape Communications, Inc., or similarly suitable browser applications.

Through browser application **552**, the user may, for example, obtain web pages pertaining to a particular entity by sending an HTTP request (e.g., GET) containing the URL (uniform resource locator) that identifies the web page file. The request sent via control connection **553** may arrive at web server **550** through the HTTP port **554**. HTTP port **554** may represent any port through which HTTP communication is enabled. HTTP port **554** may also represent the default port for communicating web pages with client browsers. The HTTP default port may represent, for example, either port **80** or port **8080** on web server **550**. As is known, one or both of these ports on web server **550** may be available for web page communication even if there are firewalls disposed between the web server **550** and client browser application **552**, which otherwise block all HTTP traffic in other ports. Using the furnished URL, web server **550** may then obtain the desired web page(s) for sending to client browser application **552** via data connection **556**.

The invention, in one embodiment, involves employing the HTTP protocol to communicate media commands from a browser application or browser plug-in to the server. Media commands are, for example, PLAY, STOP, REWIND, FAST FORWARD, and PAUSE. The server computer may represent, for example, a web server. The server computer may also represent a video server for streaming video to the client computer. Through the use of the HTTP protocol the client computer may successfully send media control requests and receive media data through any HTTP port. If the default HTTP port, e.g., port **80** or **8080**, is specified, the client may successfully send media control requests and receive media data even if there exists a firewall or an HTTP Proxy disposed in between the server computer and the client computer, which otherwise blocks all other traffic that does not use the HTTP protocol. For example, these firewalls or HTTP Proxies do not allow regular TCP or UDP packets to go through.

As is well known to those skilled, the HTTP protocol, as specified by for example the Internet Request For Comments RFC 1945 (T. Berners-Lee et al.), typically defines only three types of requests to be sent from the client computer to the server, namely GET, POST, and HEAD. The POST command, for instance, is specified in RFC 1945 to be composed of a Request-Line, one or more Headers and Entity-Body. To send media commands like PLAY, REWIND, etc., the invention in one embodiment sends the media command as part of the Entity-Body of the HTTP POST command. The media command can be in any format or protocol, and can be, for instance, in the same format as that employed when firewalls are not a concern and plain

TCP protocol can be used. This format can be, for example, RTSP (Real Time Streaming Protocol).

When a server gets an HTTP request, it answers the client with an HTTP Response. Responses are typically composed of a Status-Line, one or more headers, and an Entity-Body. In one embodiment of this invention, the response to the media commands is sent as the Entity-Body of the response to the original HTTP request that carried the media command.

FIG. 5B illustrates this use of HTTP for sending arbitrary media commands. In FIG. 5B, the plug-in application **560** within client browser application **562** may attempt to receive media data (e.g., video, audio, annotation, or the like) by first sending an HTTP request to server **564** via control connection **565**. For example, a REWIND command could be sent from the client **560** to the server **564** as an HTTP packet **570** of the form: "POST/HTTP/1.0<Entity-Body containing rewind command in any suitable media protocol>". The server can answer to this request with an HTTP response **572** of the form: "HTTP/1.0 200ok<Entity-Body containing rewind response in any suitable media protocol>".

The HTTP protocol can be also used to send media data across firewalls. The client can send a GET request to the video server, and the video server can then send the video data as the Entity-Body of the HTTP response to this GET request.

Some firewalls may be restrictive with respect to HTTP data and may permit HTTP packets to traverse only on a certain port, e.g., port **80** and/or port **8080**. FIG. 5C illustrates one such situation. In this case, the control and data communications for the various data stream, e.g., audio, video, and/or annotation associated with different rendering sessions (and different clients) may be multiplexed using conventional multiplexer code and/or circuit **506** at client application **300** prior to being sent via port **502** (which may represent, for example, HTTP port **80** or HTTP port **8080**). The inventive combined use of the HTTP protocol and of the multiplexer for transmitting media control and data is referred to as the HTTP multiplex protocol, and can be used to send this data across firewalls that only allow HTTP traffic on specific ports, e.g., port **80** or **8080**.

At server **402**, representing, for example, server **104** of FIG. 1, conventional demultiplexer code and/or circuit **508** may be employed to decode the received data packets to identify which stream the control request is associated with. Likewise, data sent from server **402** to client application **300** may be multiplexed in advance at server **402** using for example conventional multiplexer code and/or circuit **510**. The multiplexed data is then sent via port **502**. At client application **300**, the multiplexed data may be decoded via conventional demultiplexer code and/or circuit **512** to identify which stream the received data packets is associated with (audio, video, or annotation).

Multiplexing and demultiplexing at the client and/or server may be facilitated for example by the use of the Request-URL part of the Request-Line of HTTP requests. As mentioned above, the structure of HTTP requests is described in RFC **1945**. The Request-URL may, for example, identify the stream associated with the data and/or control request being transmitted. In one embodiment, the additional information in the Request-URL in the HTTP header may be as small as one or a few bits added to the HTTP request sent from client application **300** to server **402**.

To further facilitate discussion of the inventive HTTP multiplexing technique, reference may now be made to FIG. 5D. In FIG. 5D, the plug-in application **660** within client plug-in application **660** may attempt to receive media data (e.g., video, audio, annotation, or the like) by first sending a control request **670** to server **664** via control connection **665**. The control request is an HTTP request, which arrives at the HTTP default port **654** on server **664**. As mentioned earlier, the default HTTP port may be either port **80** or port **8080** in one embodiment.

In one example, the control request **670** from client plug-in **660** takes the form of a command to "POST/12469 HTTP/1.0<Entity-Body>" which indicates to the server (through the designation 12469 as the Request-URL) that this is a control connection. The Entity-Body contains, as described above, binary data that informs the video server that the client plug-in **660** wants to display a certain video or audio clip. Software codes within server **664** may be employed to assign a unique ID to this particular request from this particular client.

For discussion sake, assume that server **664** associates unique ID 35,122 with a video data connection between itself and client plug-in application **660**, and unique ID 29,999 with an audio data connection between itself and client plug-in application. The unique ID is then communicated as message **672** from server **664** to client plug-in application **660**, again through the aforementioned HTTP default port using data connection **667**. The Entity-Body of message **672** contains, among other things and as depicted in detail **673**, the audio and/or video session ID. Note that the unique ID is unique to each data connection (e.g., each of the audio, video, and annotation connections) of each client plug-in application (since there may be multiple client plug-in applications attempting to communicate through the same port).

Once the connection is established, the same unique ID number is employed by the client to issue HTTP control requests to server **664**. By way of example, client plug-in application **660** may issue a command "GET /35,122 HTTP/1.0" or "POST /35,122 HTTP/1.0<Entity-Body containing binary data with the REWIND media command>" to request a video file or to rewind on the video file. Although the rewind command is used in FIGS. **5A–5D** to facilitate ease of discussion, other media commands, e.g., fast forward, pause, real-time play, live-play, or the like, may of course be sent in the Entity-Body. Note that the unique ID is employed in place of or in addition to the Request-URL to qualify the Request-URL.

Once the command is received by server **664**, the unique ID number (e.g. 35,122) may be employed by the server to demultiplex the command to associate the command with a particular client and data file. This unique ID number can also attach to the HTTP header of HTTP responses sent from server **664** to client plug-in application **660**, through the same HTTP default port **654** on server **664**, to permit client plug-in application **660** to ascertain whether an HTTP data packet is associated with a given data stream.

Advantageously, the invention permits media control commands and media data to be communicated between the client computer and the server computer via the default HTTP port, e.g., port **80** or **8080** in one embodiment, even if HTTP packets are otherwise blocked by a firewall disposed between the client computer and the server computer. The association of each control connection and data connection to each client with a unique ID advantageously permits multiple control and data connections (from one or more clients) to be established through the same default HTTP port on the server, advantageously bypassing the firewall. Since both the server and the client have the demultiplexer code and/or circuit that resolve a particular unique ID into a particular data stream, multiplexed data communication is advantageously facilitated thereby.

In some networks, it may not be possible to traverse the firewall due to stringent firewall policies. As mentioned earlier, it may be possible in these situations to allow the client application to communicate with a server using a proxy. FIG. **6** illustrates this situation wherein client application **300** employs proxy **602** to communicate with server **402**. The use of proxy **602** may be necessary since client application **300** may employ a protocol which is strictly prohibited by firewall **604**. The identity of proxy **602** may be found in browser program **306**, e.g., Netscape as it employs the proxy to download its web pages, or may be configured by the user himself. Typical protocols that may employ a proxy for communication, e.g., proxy **602**, includes HTTP and UDP.

In accordance with one embodiment of the present invention, the multiple protocols that may be employed for communication between a server computer and a client computer are tried in parallel during autodetect. In other words, the connections depicted in FIGS. **3**, **4**, **5**C, and **6** may be attempted simultaneously and in parallel over different control connections by the client computer. Via these control connections, the server is requested to respond with various protocols.

If the predefined "best" protocol (predetermined in accordance with some predefined protocol priority) is received by the client application from the server, autodetect may, in one embodiment, end immediately and the "best" protocol is selected for immediate communication. In one real-time data rendering application, UDP is considered the "best" protocol, and the receipt of UDP data by the client may trigger the termination of the autodetect.

If the "best" protocol has not been received after a predefined time period, the most advantageous protocol (in terms of for example data transfer rate and/or transmission control) is selected among the set of protocols received by the client. The selected protocol may then be employed for communication between the client and the server.

FIG. **7** depicts, in accordance with one embodiment of the present invention, a simplified flowchart illustrating the steps of the inventive autodetect technique. In FIG. **7**, the client application starts (in step **702**) by looking up the HTTP proxy, if there is any, from the browser. As stated earlier, the client computer may have received a web page from the browser, which implies that the HTTP protocol may have been employed by the browser program for communication. If a HTTP proxy is required, the name and location of the HTTP proxy is likely known to the browser, and this knowledge may be subsequently employed by the client to at least enable communication with the server using the HTTP proxy protocol, i.e., if a more advantageous protocol cannot be ascertained after autodetect.

In step **704**, the client begins the autodetect sequence by starting in parallel the control thread **794**, along with five protocol threads **790**, **792**, **796**, **798**, and **788**. As the term is used herein, parallel refers to both the situation wherein the multiple protocol threads are sent parallelly starting at substantially the same time (having substantially similar starting time), and the situation wherein the multiple protocol threads simultaneously execute (executing at the same time), irrespective when each protocol thread is initiated. In the latter case, the multiple threads may have, for example, staggered start time and the initiation of one thread may not depend on the termination of another thread.

Control tread **794** represents the thread for selecting the most advantageous protocol for communication. The other protocol threads **790**, **792**, **796**, **798**, and **788** represent threads for initiating in parallel communication using the various protocols, e.g., UDP, TCP, HTTP proxy, HTTP through port **80** (HTTP **80**), and HTTP through port **8080** (HTTP **8080**). Although only five protocol threads are shown, any number of protocol threads may be initiated by the client, using any conventional and/or suitable protocols. The steps associated with each of threads **794**, **790**, **792**, **796**, **798**, and **788** are discussed herein in connection with FIGS. **8**A–**8**E and **9**.

In FIG. **8**A, the UDP protocol thread is executed. The client inquires in step **716** whether there requires a UDP proxy. If the UDP proxy is required, the user may obtain the name of the UDP proxy from, for example, the system administrator in order to use the UDP proxy to facilitate communication to the proxy (in step **718**). If no UDP proxy is required, the client may directly connect to the server (in step **720**). Thereafter, the client may begin sending a data request (i.e., a control request) to the server in step **722** using the UDP protocol (either through the proxy if a proxy is involved or directly to the server if no proxy is required).

In FIG. **8**B, the TCP protocol thread is executed. If TCP protocol is employed, the client typically directly connects to the server (in step **726**). Thereafter, the client may begin sending a data request (i.e., a control request) to the server using the TCP protocol (step **724**).

In FIG. **8**C, the HTTP protocol thread is executed. The client inquires in step **716** whether there requires a HTTP proxy. If the HTTP proxy is required, the user may obtain the name of the HTTP proxy from, for example, the browser since, as discussed earlier, the data pertaining to the proxy may be kept by the browser. Alternatively, the user may obtain data pertaining to the HTTP proxy from the system administrator in order to use the HTTP proxy to facilitate communication to the server (in step **732**).

If no HTTP proxy is required, the client may directly connect to the server (in step **730**). Thereafter, the client may begin sending a data request (i.e., a control request) to the server in step **734** using the HTTP protocol (either through the proxy if a proxy is involved or directly to the server if no proxy is required).

In FIG. **8**D, the HTTP **80** protocol thread is executed. If HTTP **80** protocol is employed, HTTP data may be exchanged but only through port **80**, which may be for example the port on the client computer through which communication with the network is permitted. Through port **80**, the client typically directly connects to the server (in step **736**). Thereafter, the client may begin sending a data request (i.e., a control request) to the server (step **738**) using the HTTP **80** protocol.

In FIG. **8**E, the HTTP **8080** protocol thread is executed. If HTTP **8080** protocol is employed, HTTP data may be exchanged but only through port **8080**, which may be the port on the client computer for communicating with the network. Through port **8080**, the client typically directly connects to the server (in step **740**). Thereafter, the client may begin sending a data request (i.e., a control request) to the server (step **742**) using the HTTP **8080** protocol. The multiplexing and demultiplexing techniques that may be employed for communication through port **8080**, as well as port **80** of FIG. **8**D, have been discussed earlier and are not repeated here for brevity sake.

FIG. **9** illustrates, in accordance with one embodiment of the present invention, control thread **794** of FIG. **7**. It should be emphasized that FIG. **7** is but one way of implementing the control thread; other techniques of implementing the control thread to facilitate autodetect should be apparent to those skilled in the art in view of this disclosure. In step **746**, the thread determines whether the predefined timeout period has expired. The predefined timeout period may be any predefined duration (such as 7 seconds for example) from the time the data request is sent out to the server (e.g., step **722** of FIG. **8**A). In one embodiment, each protocol thread has its own timeout period whose expiration occurs at the expiration of a predefined duration after the data request using that protocol has been sent out. When all the timeout periods associated with all the protocols have been accounted for, the timeout period for the autodetect technique is deemed expired.

If the timeout has occured, the thread moves to step **754** wherein the most advantageous protocol among the set of protocols received back from the server is selected for communication. As mentioned, the selection of the most advantageous protocol may be performed in accordance with some predefined priority scheme, and data regarding the selected protocol may be saved for future communication sessions between this server and this client.

If no timeout has occurred, the thread proceeds to step **748** to wait for either data from the server or the expiration of the timeout period. If timeout occurs, the thread moves to step **754**, which has been discussed earlier. If data is received from the server, the thread moves to step **750** to ascertain whether the protocol associated with the data received from the server is the predefined "best" protocol, e.g., in accordance with the predefined priority.

If the predefined "best" protocol (e.g., UDP in some real-time data rendering applications) is received, the thread preferably moves to step **754** to terminate the autodetect and to immediately begin using this protocol for data communication instead of waiting of the timeout expiration. Advantageously, the duration of the autodetect sequence may be substantially shorter than the predefined timeout period. In this manner, rapid autodetect of the most suitable protocol and rapid establishment of communication are advantageously facilitated.

If the predefined "best" protocol is not received in step **750**, the thread proceeds to step **752** to add the received protocol to the received set. This received protocol set represents the set of protocols from which the "most advantageous" (relatively speaking) protocol is selected. The most advantageous protocol is ascertained relative to other protocols in the received protocol set irrespective whether it is the predefined "best" protocol in accordance with the predefined priority. As an example of a predefined protocol priority, UDP may be deemed to be best (i.e., the predefined best), followed by TCP, HTTP, then HTTP **80** and HTTP **8080** (the last two may be equal in priority). As mentioned earlier, the most advantageous protocol is selected from the received protocol set preferably upon the expiration of the predefined timeout period.

From step **752**, the thread returns to step **746** to test whether the timeout period has expired. If not, the thread continues along the steps discussed earlier.

Note that since the invention attempts to establish communication between the client application and the server computer in parallel, the time lag between the time the autodetect mechanism begins to execute and the time when the most advantageous protocol is determined is minimal. If communication attempts have been tried in serial, for example, the user would suffer the delay associated with each protocol thread in series, thereby disadvantageously lengthening the time period between communication attempt and successful establishment of communication.

The saving in time is even more dramatic in the event the network is congested or damaged. In some networks, it may take anywhere from 30 to 90 seconds before the client application realizes that an attempt to connect to the server (e.g., step **720**, **726**, **730**, **736**, or **740**) has failed. If each protocol is tried in series, as is done in one embodiment, the delay may, in some cases, reach minutes before the user realizes that the network is unusable and attempts should be made at a later time.

By attempting to establish communication via the multiple protocols in parallel, network-related delays are suffered in parallel. Accordingly, the user does not have to wait for multiple attempts and failures before being able to ascertain that the network is unusable and an attempt to establish communication should be made at a later time. In one embodiment, once the user realizes that all parallel attempts to connect with the network and/or the proxies have failed, there is no need to make the user wait until the expiration of the timeout periods of each thread. In accordance with this embodiment, the user is advised to try again as soon as it is realized that parallel attempts to connect with the server have all failed. In this manner, less of the user's time is needed to establish optimal communication with a network.

While this invention has been described in terms of several preferred embodiments, there are alterations, permutations, and equivalents which fall within the scope of this invention. For example, although the invention has been described with reference with sending out protocol threads in parallel, the automatic protocol detection technique also applies when the protocol threads are sent serially. In this case, while it may take longer to select the most advantageous protocol for selection, the automatic protocol detection technique accomplishes the task without requiring any sophisticated technical knowledge on the part of the user of the client computer. The duration of the autodetect technique, even when serial autodetect is employed, may be shortened by trying the protocols in order of their desirability and ignoring less desirable protocols once a more desirable protocol is obtained. It should also be noted that there are many alternative ways of implementing the methods and apparatuses of the present invention. It is therefore intended that the following appended claims be interpreted as including all such alterations, permutations, and equivalents as fall within the true spirit and scope of the present invention.

What is claimed is:

1. In a computer network, a method for automatically detecting a most advantageous protocol for communication by a client computer, said client computer being configured to be coupled to a server computer via a computer network, the method comprising:

sending, from said client computer to said server computer, a plurality of data requests substantially in parallel, each of said data requests employing a different protocol and a different connection, said data requests being configured to solicit, responsive to said data requests, a set of responses from said server computer, each of said responses employing a protocol associated with a respective one of said data requests;

receiving at said client computer at least a subset of said responses; monitoring said subset of said responses as each response is received from said server computer to select said most advantageous protocol from protocols associated with said subset of said responses, wherein said most advantageous protocol is determined based on a predefined protocol priority that designates a predefined best protocol; and

in response to expiration of a timeout period without receiving a response employed by the predefined best protocol, selecting the most advantageous protocol from the protocols employed by the responses received at the client computer, the selection based at least in part on at least one of a data transfer rate and transmission control characteristics, said timeout period being measured from a transmitting time of a data request.

2. The method of claim **1** wherein said plurality of data requests are sent substantially at the same time.

3. The method of claim **1** wherein said plurality of data requests execute concurrently.

4. The method of claim **1** wherein said client computer, upon receiving a response employing said predefined best protocol, immediately designates said predefined best protocol as said most advantageous protocol.

5. The method of claim **1** wherein at least one data request is sent using a multiplexed HTTP protocol.

6. The method of claim **1** wherein said client computer is configured and arranged to execute an application for rendering real-time data as said real-time data is received from said server computer.

7. The method of claim **1** wherein said computer network comprises the Internet and said predefined best protocol is UDP.

8. The method of claim **7** wherein said real-time data represents one of a video data stream, an audio data stream, and an annotation data stream.

9. In a computer network, a method for automatically detecting a most advantageous protocol for communication by a client computer, said client computer being configured to be coupled to a server computer via a computer network, the method comprising:

sending, from said client computer to said server computer, a plurality of data requests substantially in parallel, each of said data requests employing a different protocol and a different connection;

receiving at least a subset of said data requests at said server computer;

sending a set of responses from said server computer to said client computer, said set of responses being responsive to said subset of said data requests, each of said responses employing a protocol associated with a respective one of said subset of said data requests;

receiving at said client computer at least a subset of said responses; and

selecting, for said communication between said client computer and said server computer, said most advantageous protocol from protocols associated with said subset of said responses, wherein said most advantageous protocol is determined based on a predefined protocol priority that designates a predefined best protocol if a response employing the predefined best protocol is received at the client computer, and wherein, if a timeout period expires without the client computer receiving a response employing the predefined best protocol, the most advantageous protocol is selected from the protocols employed by the responses received at the client computer based at least in part on at least one of a data transfer rate and transmission control characteristics, said timeout period being measured from a transmitting time of a data request.

10. The method of claim **9** wherein selecting the most advantageous protocol comprises:

monitoring, employing said client computer, said subset of said responses as each one of said subset of said responses is received at said client computer from said server computer for a response employing said predefined best protocol; and

designating said predefined best protocol said most advantageous protocol, thereby immediately permitting said client computer to employ said predefined best protocol for communication with said server computer without further receiving additional responses from said server computer.

11. The method of claim **9** wherein said computer network comprises the Internet and said predefined best protocol is UDP.

12. The method of claim **9** wherein said client computer is configured and arranged to execute an application for rendering real-time data as said real-time data is received from said server computer.

13. The method of claim **12** wherein said real-time data represents one of a video data stream, an audio data stream, and an annotation data stream.

14. The method of claim **13** wherein said data requests employ at least one of a UDP, TCP, HTTP proxy, HTTP **80**, and HTTP **8080** protocols.

15. The method of claim **14** wherein said HTTP **80** protocol is a protocol for permitting multiple HTTP data streams to be transmitted in a multiplexed manner through port **80**, said multiple HTTP data streams representing said video data stream and said audio data stream.

16. The method of claim **14** wherein said HTTP **8080** protocol is a protocol for permitting multiple HTTP data streams to be transmitted in a multiplexed manner through port **8080**, said multiple HTTP data streams representing said video data stream and said audio data stream.

17. A computer-readable medium containing computer-readable instructions for automatically detecting a most advantageous protocol for communication by a client computer, said client computer being configured for coupling to a server computer via a computer network, said computer-readable instructions comprise:

computer-readable instructions for sending in a substantially parallel manner, from said client computer to said server computer, a plurality of data requests, each of said data requests employing a different protocol and a different connection, said data requests being configured to solicit, responsive to said data requests, a set of responses from said server computer, each of said responses employing a protocol associated with a respective one of said data requests;

computer-readable instructions for receiving at said client computer at least a subset of said responses;

computer-readable instructions for monitoring said subset of said responses as each response is received from said server computer to select said most advantageous protocol from protocols associated with said subset of said responses, wherein said most advantageous protocol is determined based on a predefined protocol priority that designates a predefined best protocol; and

computer-readable instructions for, in response to expiration of a timeout period without receiving a response employing the predefined best protocol, selecting the most advantageous protocol from the protocols employed by the responses received at the client computer, the selection based at least in part on at least one of a data transfer rate and transmission control characteristics, said timeout period being measured from a transmitting time of a data request.

18. The computer-readable medium of claim **17** wherein said client computer, upon receiving a response employing said predefined best protocol, immediately designates said predefined best protocol said most advantageous protocol.

19. The computer-readable medium of claim **17** wherein at least one data request is sent using a multiplexed HTTP protocol.

20. The computer-readable medium of claim **17** wherein said client computer is configured and arranged to execute an application for rendering real-time data as said real-time data is received from said server computer.

21. The computer-readable medium of claim **17** wherein said computer network comprises the Internet and said predefined best protocol is UDP.

22. The computer readable medium of claim **21** wherein said real-time data represents one of a video data stream, an audio data stream, and an annotation data stream.

23. A computer-readable medium containing computer-readable instructions for automatically detecting a most advantageous protocol for communication by a client computer, said client computer being, configured for coupling to a server computer via a computer network, said computer-readable instructions comprise:

computer-readable instructions for sending from said client computer to said server computer, a plurality of data requests substantially in parallel, each of said data requests employing a different protocol and a different connection;

computer-readable instructions for receiving, at least a subset of said data requests at said server computer;

computer-readable instructions for sending a set of responses from said server computer to said client computer, said set of responses being responsive to said subset of said data requests, each of said responses employing a protocol associated with a respective one of said subset of said data requests;

computer-readable instructions for receiving at said client computer at least a subset of said responses; and

computer-readable instructions for selecting, for said communication between said client computer and said server computer, said most advantageous protocol from protocols associated with said subset of said responses, wherein said most advantageous protocol is determined based on a predefined protocol priority that designates a predefined best protocol if a response employing the predefined best protocol is received at the client computer, and wherein, if a timeout period expires without the client computer receiving a response employing the predefined best protocol, the most advantageous protocol is selected from the protocols employed by the responses received at the client computer based at least in part on at least one of a data transfer rate and transmission control characteristics, said timeout period being measured from a transmitting time of a data request.

* * * * *

US005870549A

United States Patent [19]

Bobo, II

[11] **Patent Number: 5,870,549**

[45] **Date of Patent: *Feb. 9, 1999**

[54] **SYSTEMS AND METHODS FOR STORING, DELIVERING, AND MANAGING MESSAGES**

[76] Inventor: **Charles R. Bobo, II**, 569 Elmwood Dr., NE., Atlanta, Ga. 30306

[*] Notice: The term of this patent shall not extend beyond the expiration date of Pat. No. 5,675,507.

[21] Appl. No.: **944,741**

[22] Filed: **Oct. 6, 1997**

Related U.S. Application Data

[63] Continuation-in-part of Ser. No. 431,716, Apr. 28, 1995, Pat. No. 5,675,507.

[51] **Int. Cl.**^{6} **H04N 1/413**

[52] **U.S. Cl.** **395/200.36**; 348/14; 348/17; 358/400; 358/402

[58] **Field of Search** 395/200.36, 154; 348/17, 14; 358/400, 402, 403; 340/311.1

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,106,060	8/1978	Chapman, Jr. .
4,918,722	4/1990	Duehren et al. .
5,033,079	7/1991	Catron et al. .
5,065,427	11/1991	Godbole .
5,068,888	11/1991	Scherk et al. .
5,091,790	2/1992	Silverberg .
5,115,326	5/1992	Burgess et al. .
5,175,762	12/1992	Kochis et al. .
5,247,591	9/1993	Baran .
5,255,312	10/1993	Koshiishi .
5,257,112	10/1993	Okada .
5,291,302	3/1994	Gordon et al. .
5,291,546	3/1994	Giler et al. .
5,317,628	5/1994	Misholi et al. .
5,333,266	7/1994	Boaz et al. .
5,349,636	9/1994	Irribarren .
5,479,411	12/1995	Klein .
5,483,580	1/1996	Brandman et al. .
5,497,373	3/1996	Hulen et al. .
5,526,353	6/1996	Henley et al. .
5,608,786	3/1997	Gordon .
5,675,507	10/1997	Bobo, II .

FOREIGN PATENT DOCUMENTS

WO 96/34341	10/1996	WIPO .

OTHER PUBLICATIONS

Delrina Advertisement, 1994.

"Working with . . . Fax Mailbox" PCToday by Jim Cope (Sep. 1994, vol. 8, Issue 9).

Voice/Fax Combos by Stuart Warren, *Computer Telephony*, Sep./Oct. 1994, p. 88.

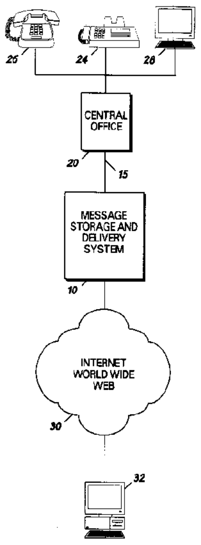
Primary Examiner—Thomas Peeso

Attorney, Agent, or Firm—Geoff L. Sutcliffe; Kilpatrick Stockton LLP

[57] **ABSTRACT**

A Message Storage and Deliver System (MSDS) is connected to the public switched telephone network (PSTN) and receives incoming calls with these calls being facsimile, voice, or data transmissions. The MSDS detects the type of call and stores the message signal in a database. The MSDS is also connected to the Internet and has a hyper-text transfer protocol deamon (HTTPD) for receiving requests from users. The HTTPD forwards requests for certain files or messages to a network server which transmits at least part of the message to the HTTPD and then to the user. In addition to requests for certain documents, the HTTPD may also receive a request in the form of a search query. The search query is forwarded from the HTTPD to an application program for conducting the search of the database. The results of the search are forwarded through the HTTPD to the user. The user may then select one or more files or messages from the search results and may save the search for later reference.

4 Claims, 18 Drawing Sheets



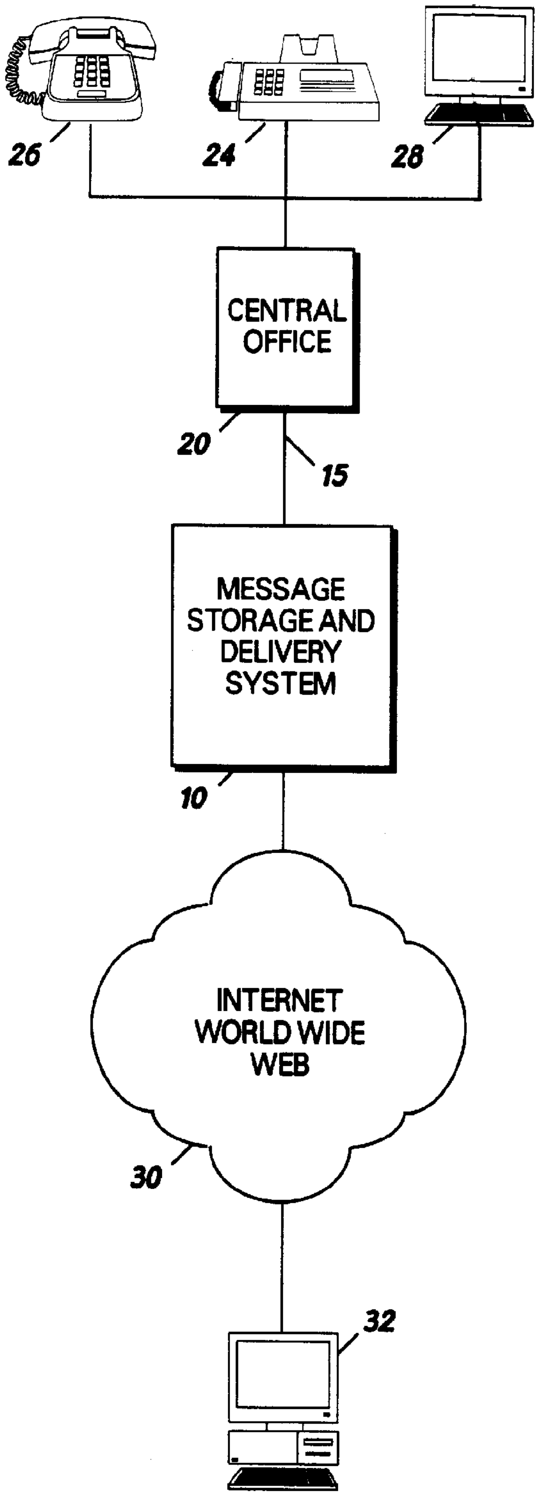

26
24
28
CENTRAL OFFICE
20
15
MESSAGE STORAGE AND DELIVERY SYSTEM
10
INTERNET WORLD WIDE WEB
30
32

FIG 1

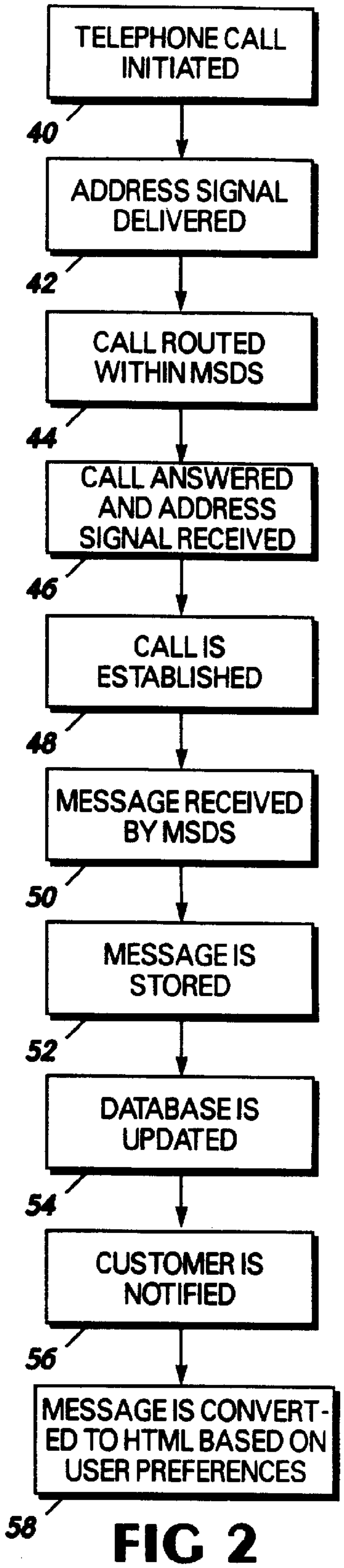

TELEPHONE CALL INITIATED
40
ADDRESS SIGNAL DELIVERED
42
CALL ROUTED WITHIN MSDS
44
CALL ANSWERED AND ADDRESS SIGNAL RECEIVED
46
CALL IS ESTABLISHED
48
MESSAGE RECEIVED BY MSDS
50
MESSAGE IS STORED
52
DATABASE IS UPDATED
54
CUSTOMER IS NOTIFIED
56
MESSAGE IS CONVERT-ED TO HTML BASED ON USER PREFERENCES
58

FIG 2

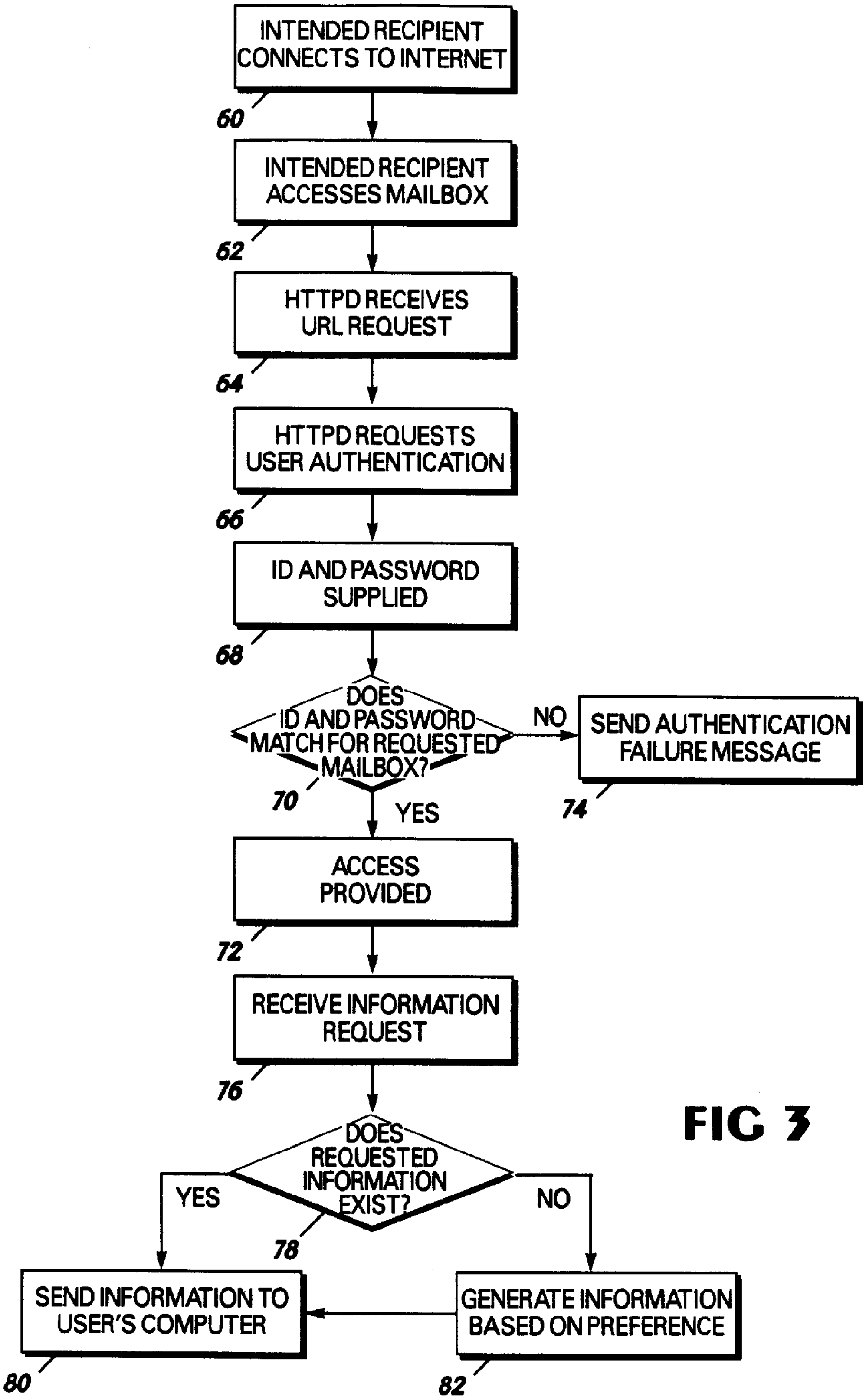

INTENDED RECIPIENT CONNECTS TO INTERNET
60
INTENDED RECIPIENT ACCESSES MAILBOX
62
HTTPD RECEIVES URL REQUEST
64
HTTPD REQUESTS USER AUTHENTICATION
66
ID AND PASSWORD SUPPLIED
68
DOES ID AND PASSWORD MATCH FOR REQUESTED MAILBOX?
NO
SEND AUTHENTICATION FAILURE MESSAGE
70
YES
74
ACCESS PROVIDED
72
RECEIVE INFORMATION REQUEST
76
DOES REQUESTED INFORMATION EXIST?
YES
NO
78
SEND INFORMATION TO USER'S COMPUTER
GENERATE INFORMATION BASED ON PREFERENCE
80
82

FIG 3

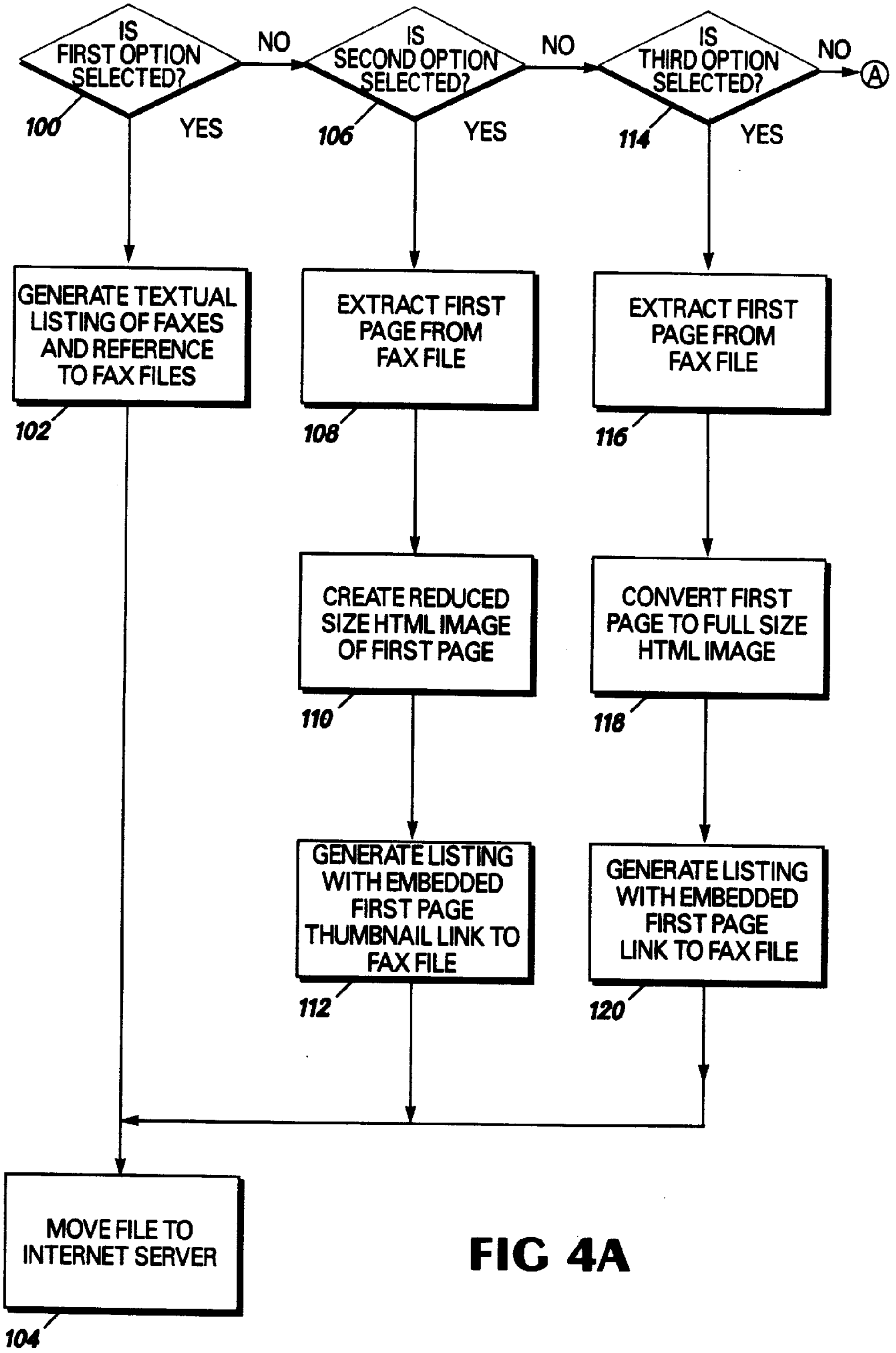

IS FIRST OPTION SELECTED?
100
YES
NO
IS SECOND OPTION SELECTED?
106
YES
NO
IS THIRD OPTION SELECTED?
114
YES
NO
A
GENERATE TEXTUAL LISTING OF FAXES AND REFERENCE TO FAX FILES
102
EXTRACT FIRST PAGE FROM FAX FILE
108
EXTRACT FIRST PAGE FROM FAX FILE
116
CREATE REDUCED SIZE HTML IMAGE OF FIRST PAGE
110
CONVERT FIRST PAGE TO FULL SIZE HTML IMAGE
118
GENERATE LISTING WITH EMBEDDED FIRST PAGE THUMBNAIL LINK TO FAX FILE
112
GENERATE LISTING WITH EMBEDDED FIRST PAGE LINK TO FAX FILE
120
MOVE FILE TO INTERNET SERVER
104

FIG 4A

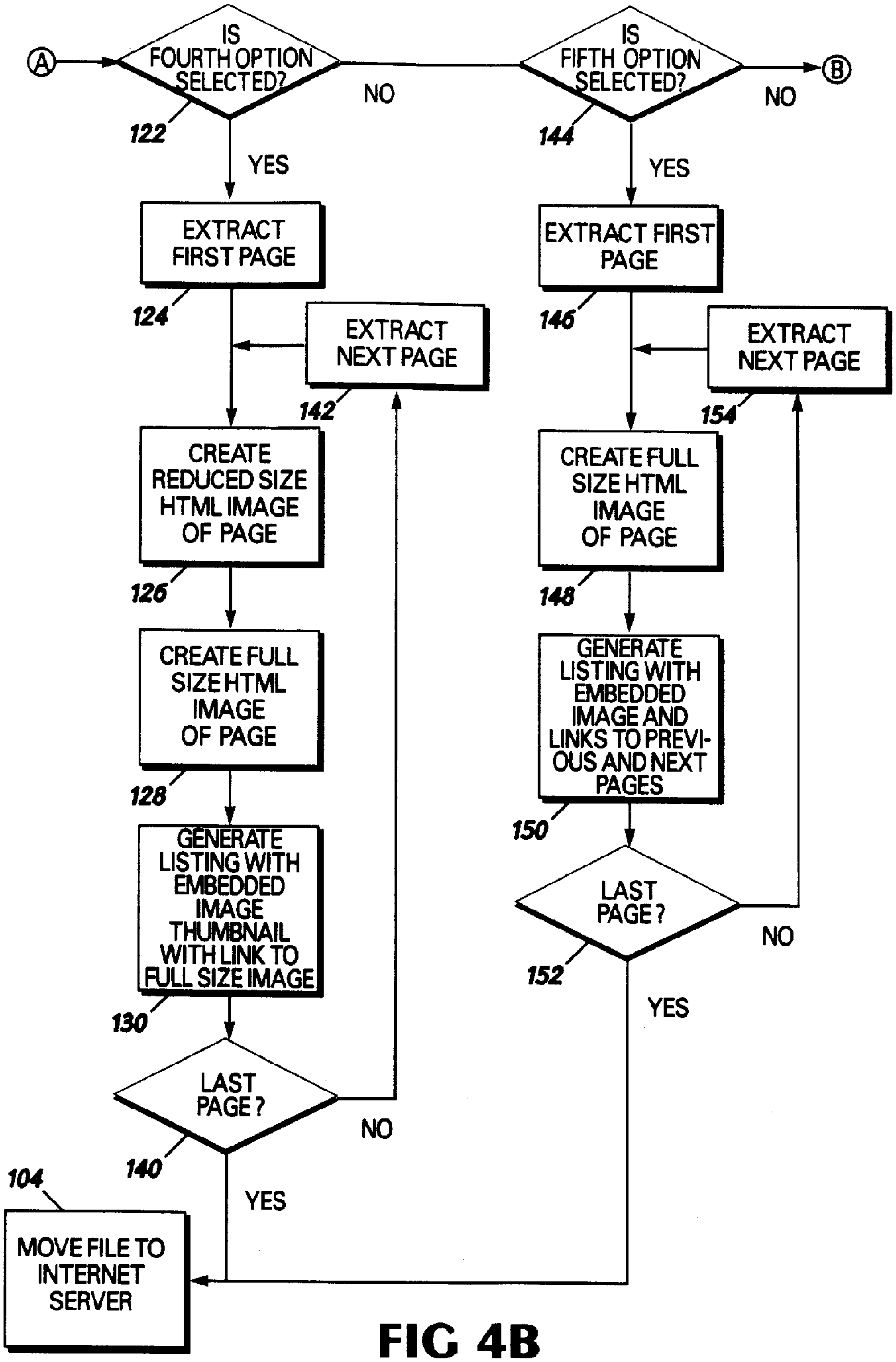

A
IS FOURTH OPTION SELECTED?
122
NO
IS FIFTH OPTION SELECTED?
144
NO
B
YES
EXTRACT FIRST PAGE
124
EXTRACT NEXT PAGE
142
CREATE REDUCED SIZE HTML IMAGE OF PAGE
126
CREATE FULL SIZE HTML IMAGE OF PAGE
128
GENERATE LISTING WITH EMBEDDED IMAGE THUMBNAIL WITH LINK TO FULL SIZE IMAGE
130
LAST PAGE ?
140
NO
YES
104
MOVE FILE TO INTERNET SERVER
YES
EXTRACT FIRST PAGE
146
EXTRACT NEXT PAGE
154
CREATE FULL SIZE HTML IMAGE OF PAGE
148
GENERATE LISTING WITH EMBEDDED IMAGE AND LINKS TO PREVI-OUS AND NEXT PAGES
150
LAST PAGE ?
152
NO
YES

FIG 4B

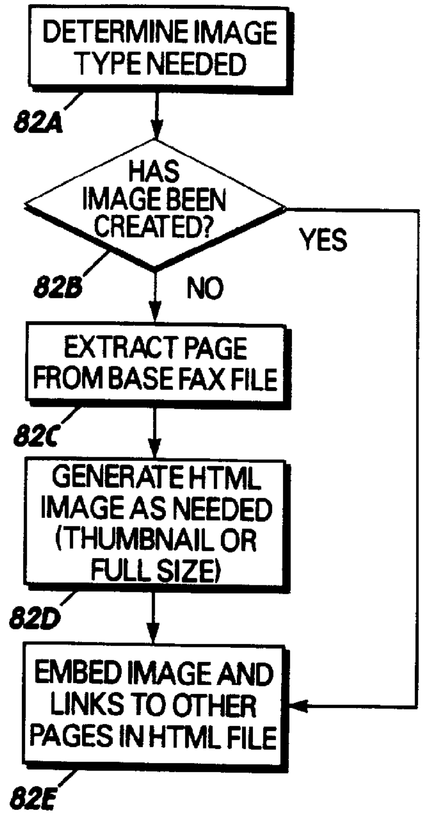

DETERMINE IMAGE TYPE NEEDED
82A
HAS IMAGE BEEN CREATED?
YES
82B
NO
EXTRACT PAGE FROM BASE FAX FILE
82C
GENERATE HTML IMAGE AS NEEDED (THUMBNAIL OR FULL SIZE)
82D
EMBED IMAGE AND LINKS TO OTHER PAGES IN HTML FILE
82E

FIG 5

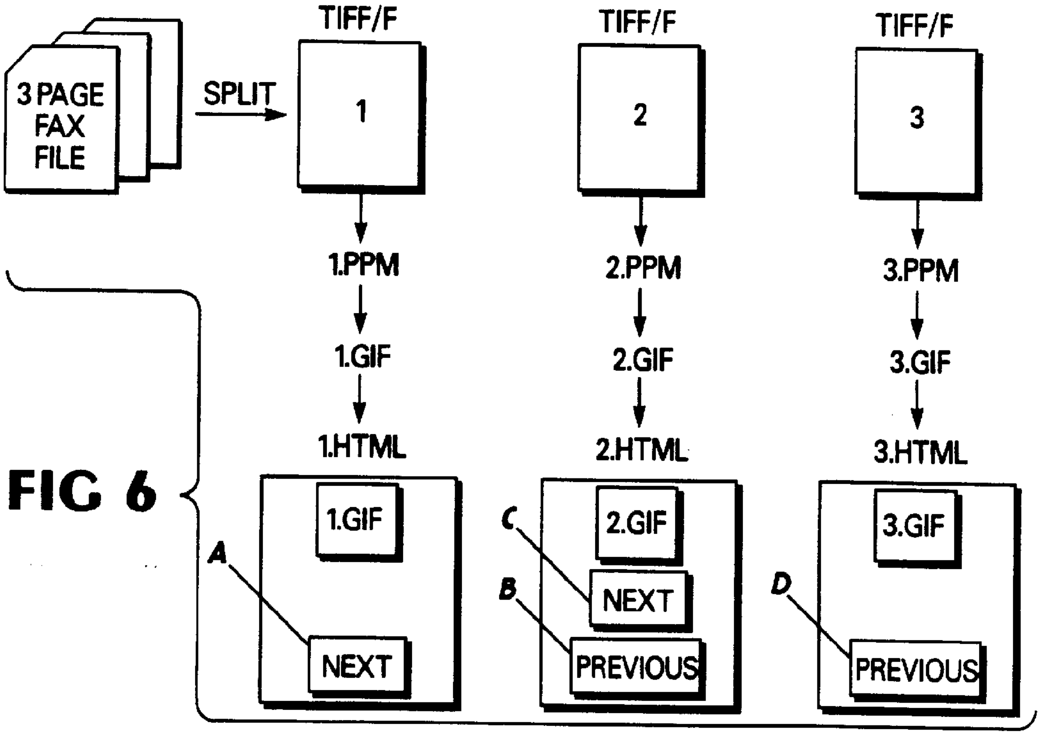
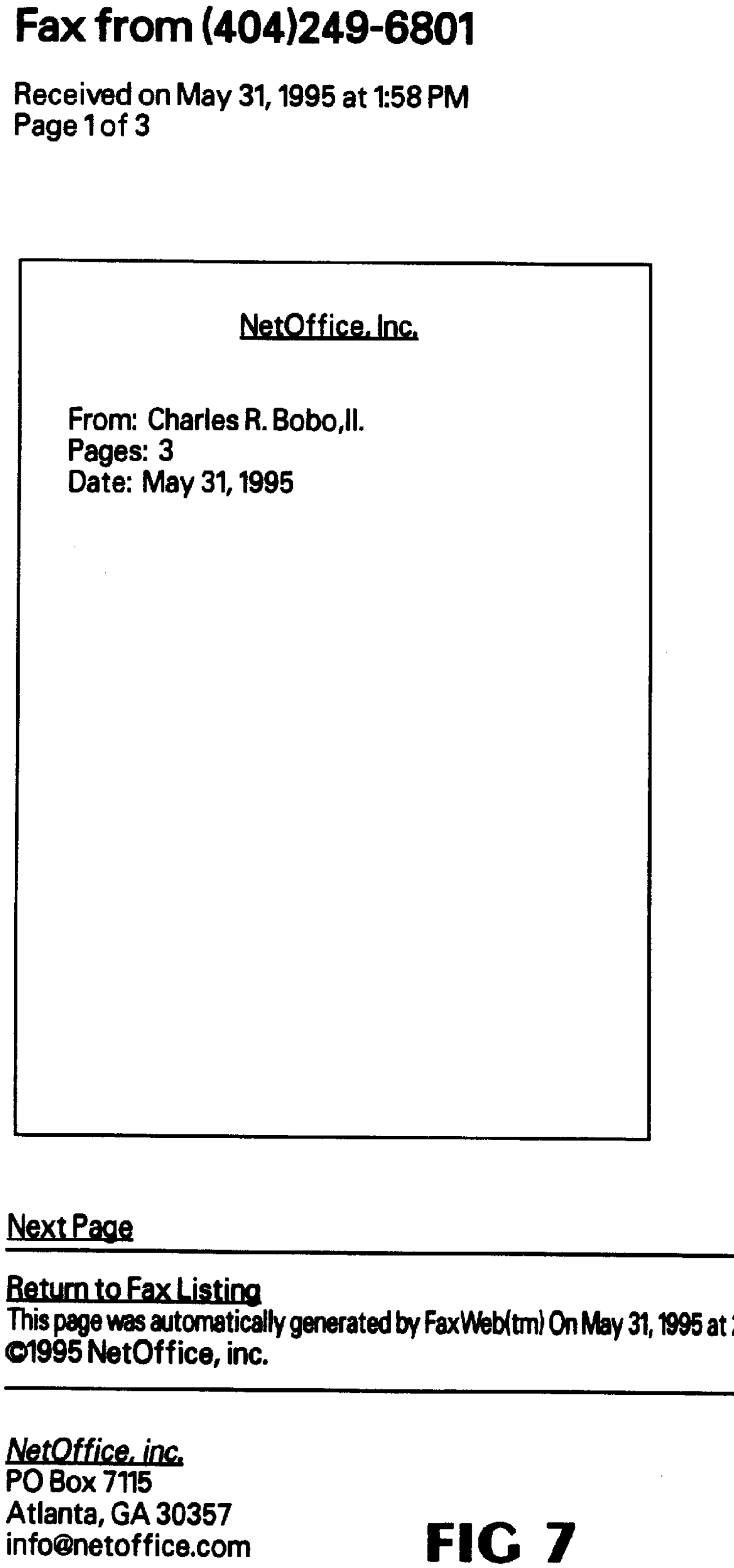

3 PAGE FAX FILE
SPLIT
TIFF/F
1
TIFF/F
2
TIFF/F
3
1.PPM
2.PPM
3.PPM
1.GIF
2.GIF
3.GIF
1.HTML
2.HTML
3.HTML
1.GIF
A
NEXT
2.GIF
C
NEXT
B
PREVIOUS
3.GIF
D
PREVIOUS

FIG 6



Fax from (404)249-6801

Received on May 31, 1995 at 1:58 PM
Page 1 of 3

NetOffice, Inc.

From: Charles R. Bobo,II.
Pages: 3
Date: May 31, 1995

Next Page

Return to Fax Listing
This page was automatically generated by FaxWeb(tm) On May 31, 1995 at 2:05pm.
©1995 NetOffice, inc.

NetOffice, inc.
PO Box 7115
Atlanta, GA 30357
info@netoffice.com

FIG 7

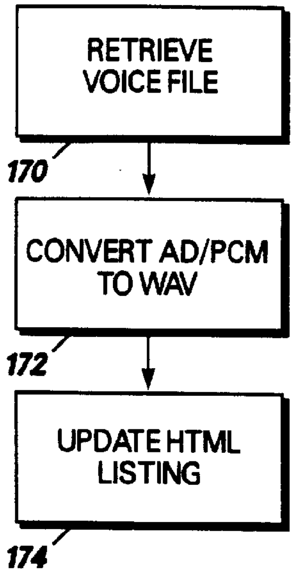


FIG 8

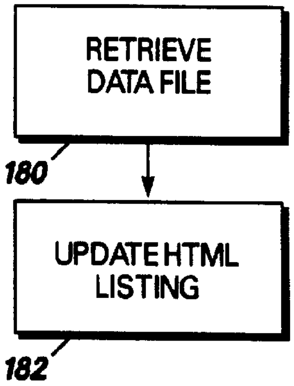


FIG 9

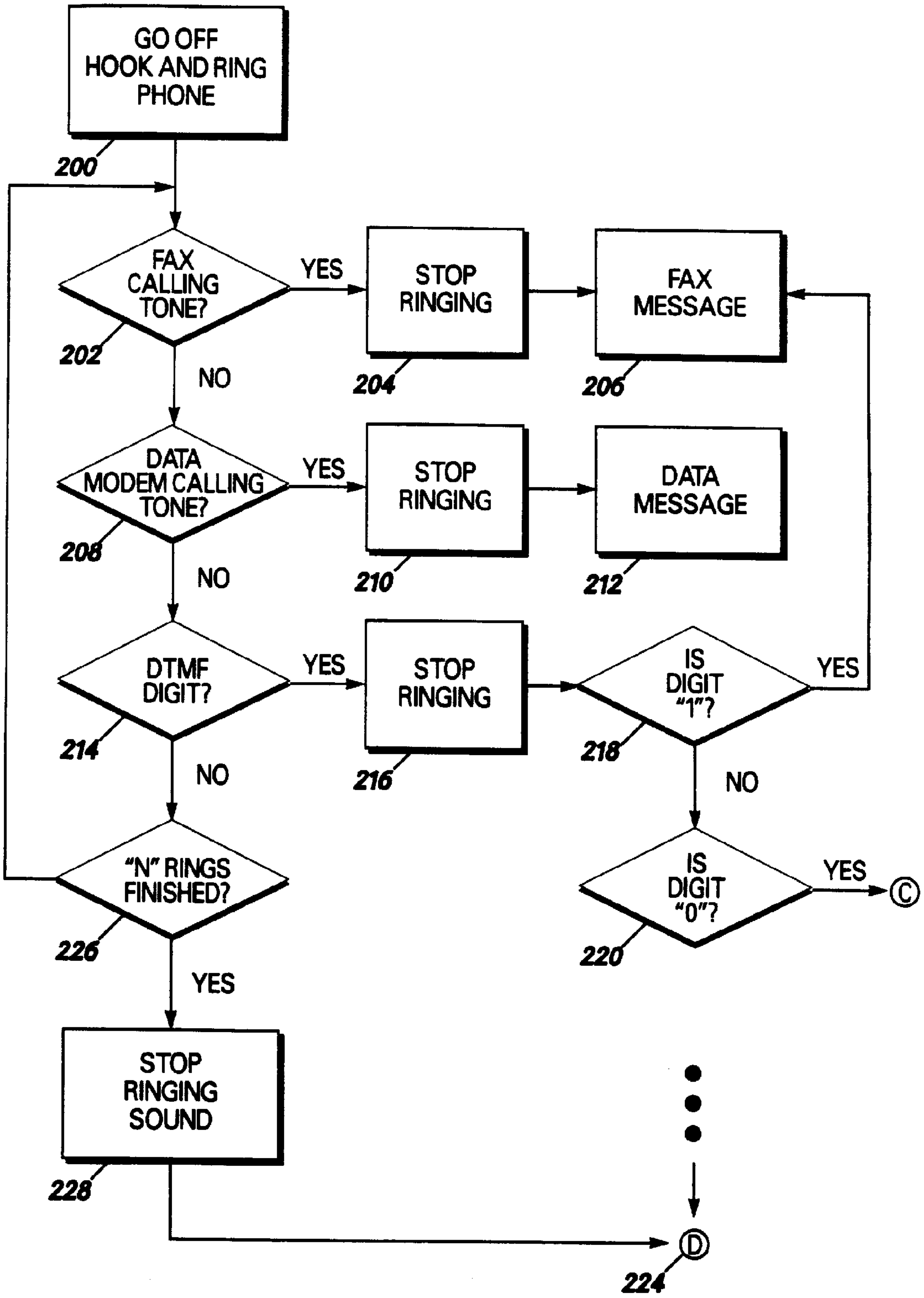

GO OFF HOOK AND RING PHONE
200
FAX CALLING TONE?
202
YES
STOP RINGING
204
FAX MESSAGE
206
NO
DATA MODEM CALLING TONE?
208
YES
STOP RINGING
210
DATA MESSAGE
212
NO
DTMF DIGIT?
214
YES
STOP RINGING
216
IS DIGIT "1"?
218
YES
NO
IS DIGIT "0"?
220
YES
C
NO
"N" RINGS FINISHED?
226
YES
STOP RINGING SOUND
228
D
224

FIG 10

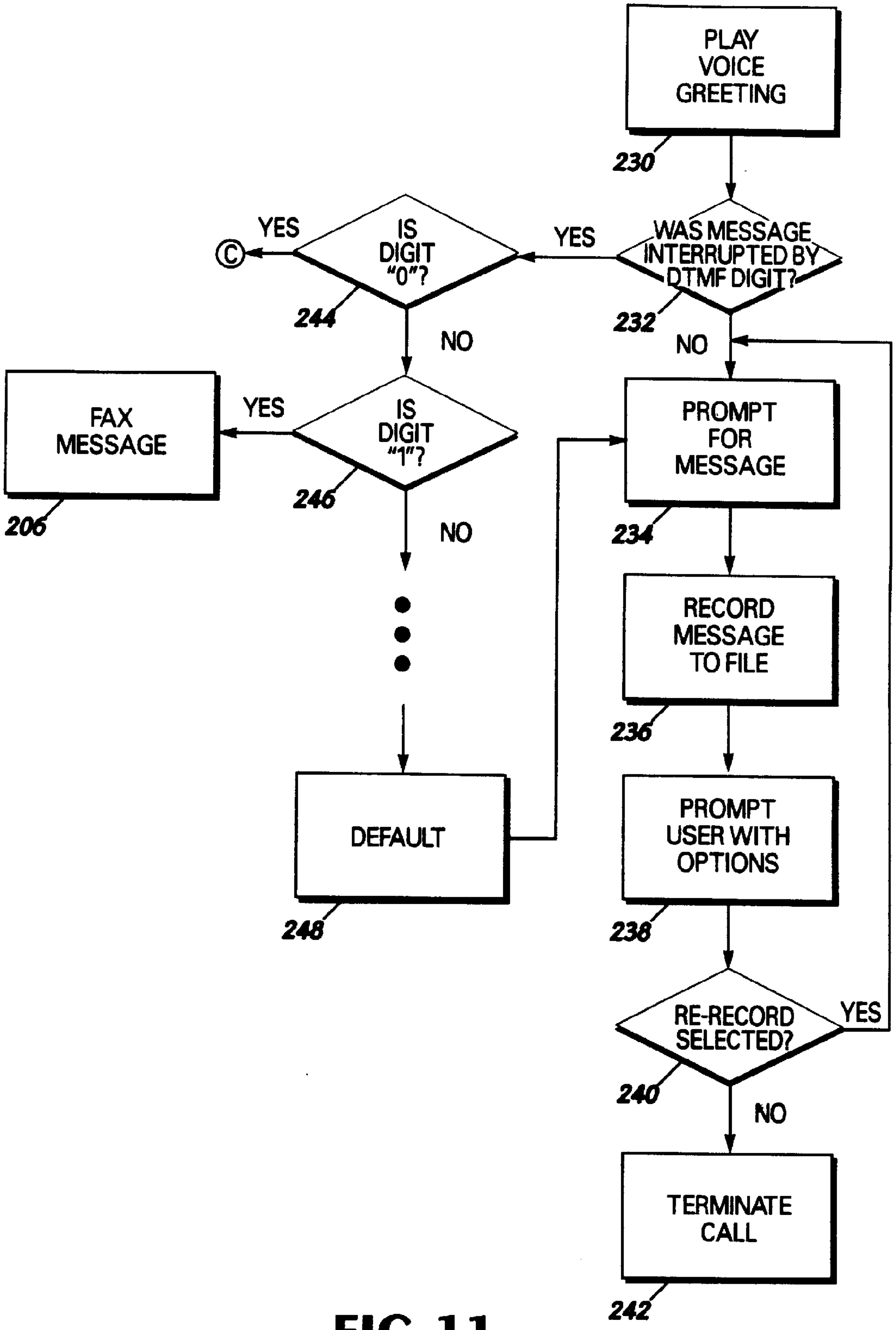

PLAY VOICE GREETING
230
WAS MESSAGE INTERRUPTED BY DTMF DIGIT?
232
YES
IS DIGIT "0"?
244
YES
C
NO
IS DIGIT "1"?
246
YES
FAX MESSAGE
206
NO
DEFAULT
248
NO
PROMPT FOR MESSAGE
234
RECORD MESSAGE TO FILE
236
PROMPT USER WITH OPTIONS
238
RE-RECORD SELECTED?
240
YES
NO
TERMINATE CALL
242

FIG 11

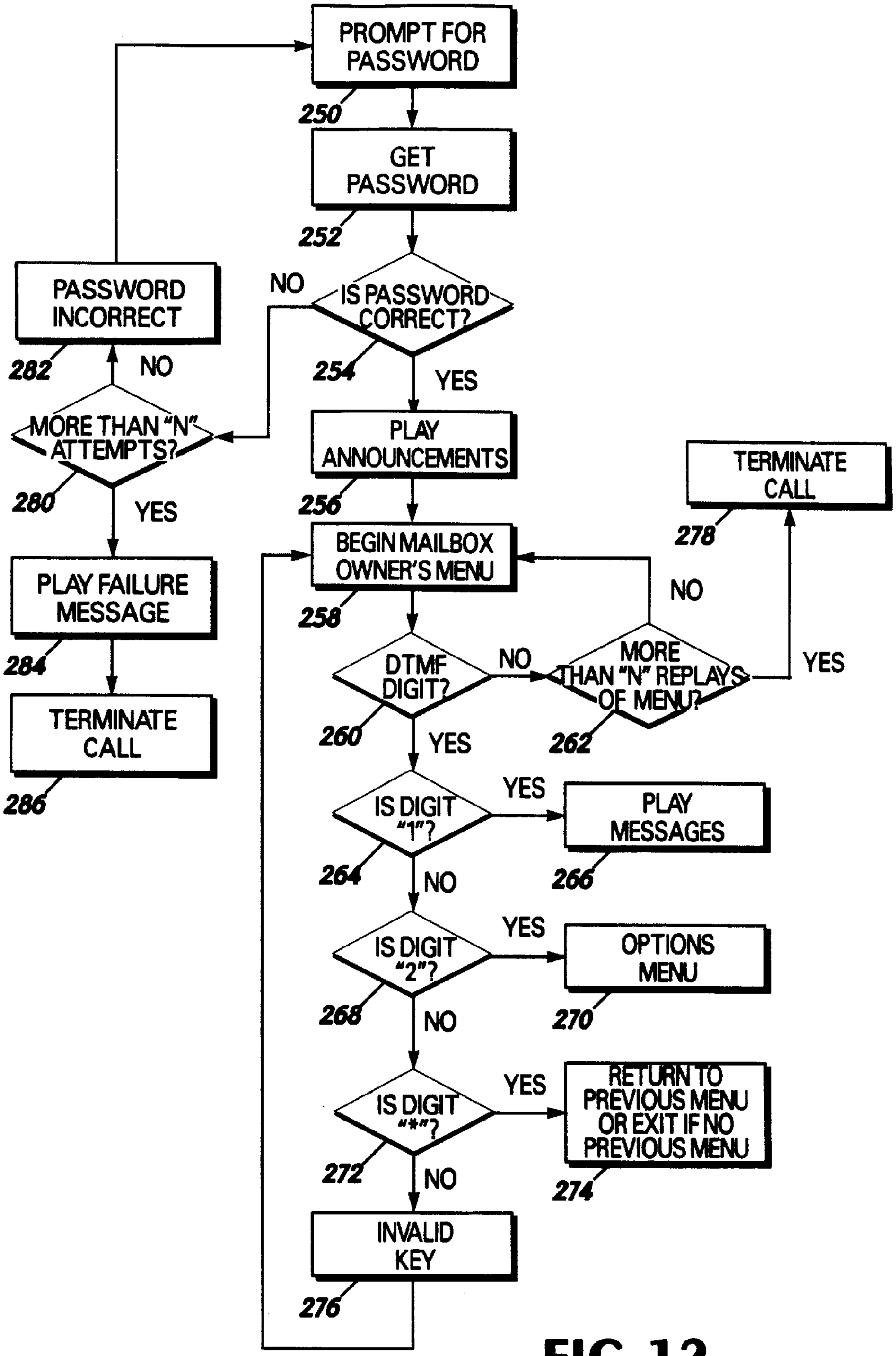

PROMPT FOR PASSWORD
250
GET PASSWORD
252
IS PASSWORD CORRECT?
254
NO
YES
PASSWORD INCORRECT
282
NO
MORE THAN "N" ATTEMPTS?
280
YES
PLAY FAILURE MESSAGE
284
TERMINATE CALL
286
PLAY ANNOUNCEMENTS
256
BEGIN MAILBOX OWNER'S MENU
258
DTMF DIGIT?
260
NO
MORE THAN "N" REPLAYS OF MENU?
262
NO
YES
TERMINATE CALL
278
YES
IS DIGIT "1"?
264
YES
PLAY MESSAGES
266
NO
IS DIGIT "2"?
268
YES
OPTIONS MENU
270
NO
IS DIGIT "*"?
272
YES
RETURN TO PREVIOUS MENU OR EXIT IF NO PREVIOUS MENU
274
NO
INVALID KEY
276

FIG 12

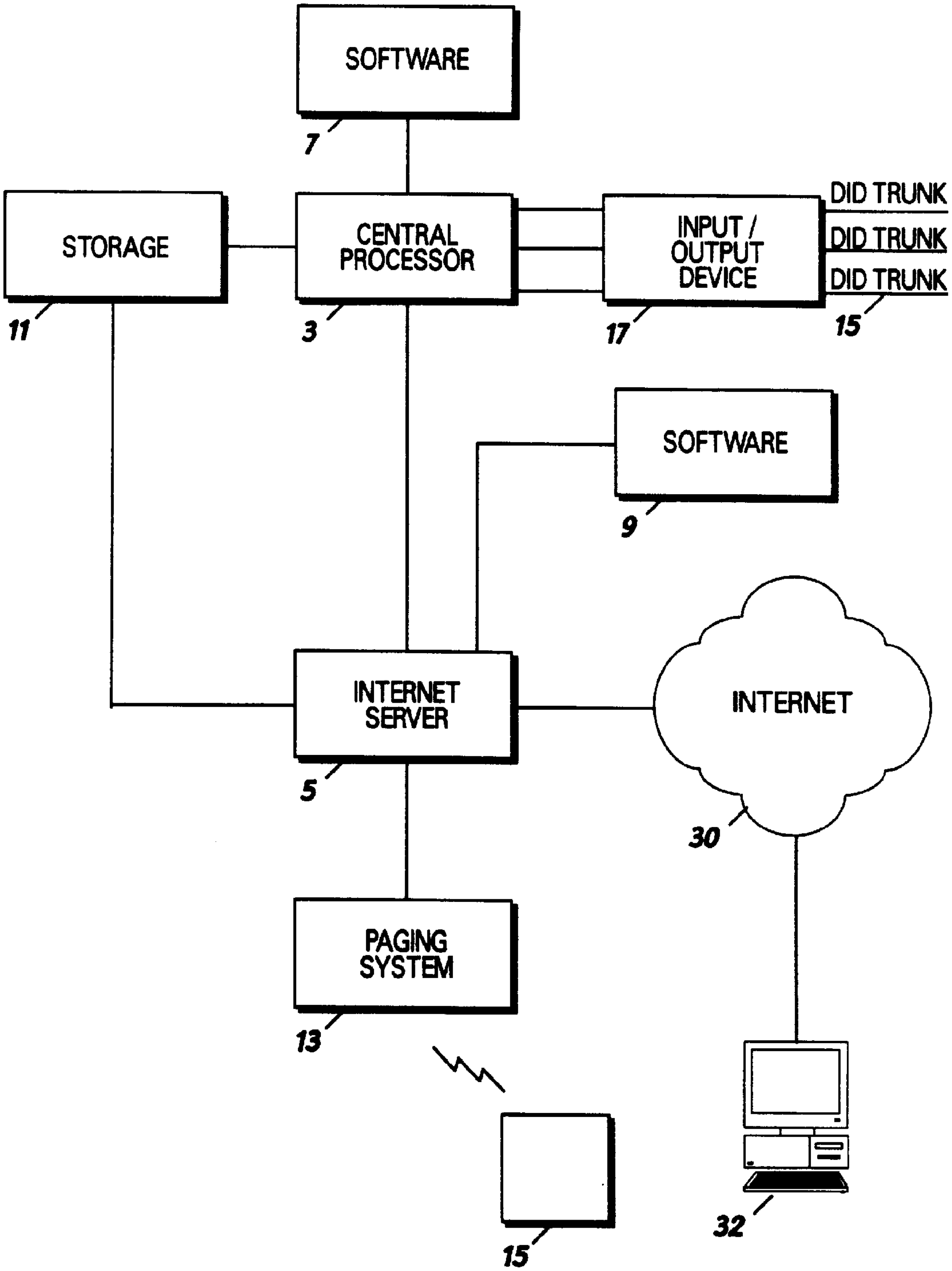

SOFTWARE
7
STORAGE
11
CENTRAL PROCESSOR
3
INPUT / OUTPUT DEVICE
17
DID TRUNK
DID TRUNK
DID TRUNK
15
SOFTWARE
9
INTERNET SERVER
5
INTERNET
30
PAGING SYSTEM
13
15
32

FIG 13

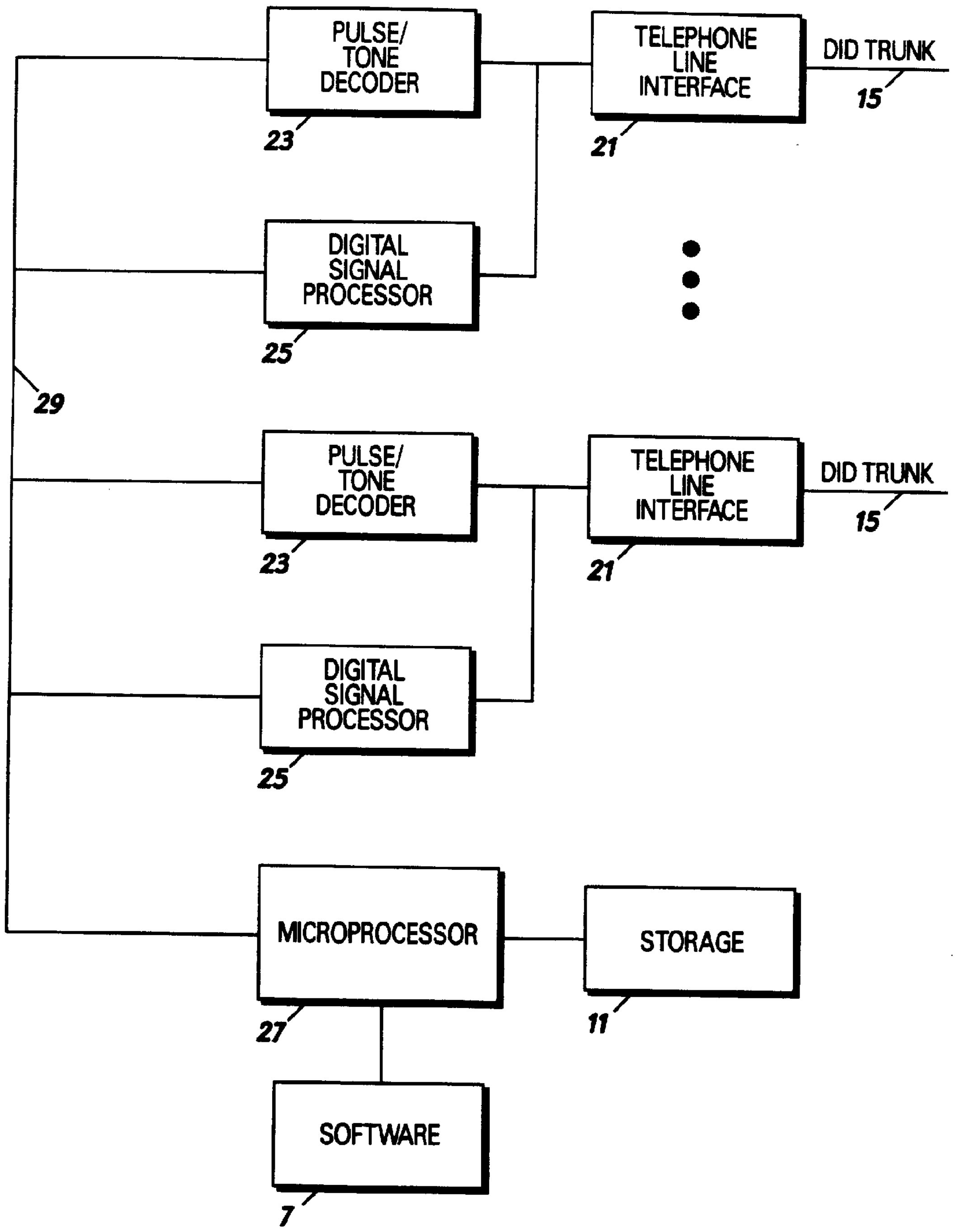

PULSE/ TONE DECODER
23
TELEPHONE LINE INTERFACE
21
DID TRUNK
15
DIGITAL SIGNAL PROCESSOR
25
29
PULSE/ TONE DECODER
23
TELEPHONE LINE INTERFACE
21
DID TRUNK
15
DIGITAL SIGNAL PROCESSOR
25
MICROPROCESSOR
27
STORAGE
11
SOFTWARE
7

FIG 14

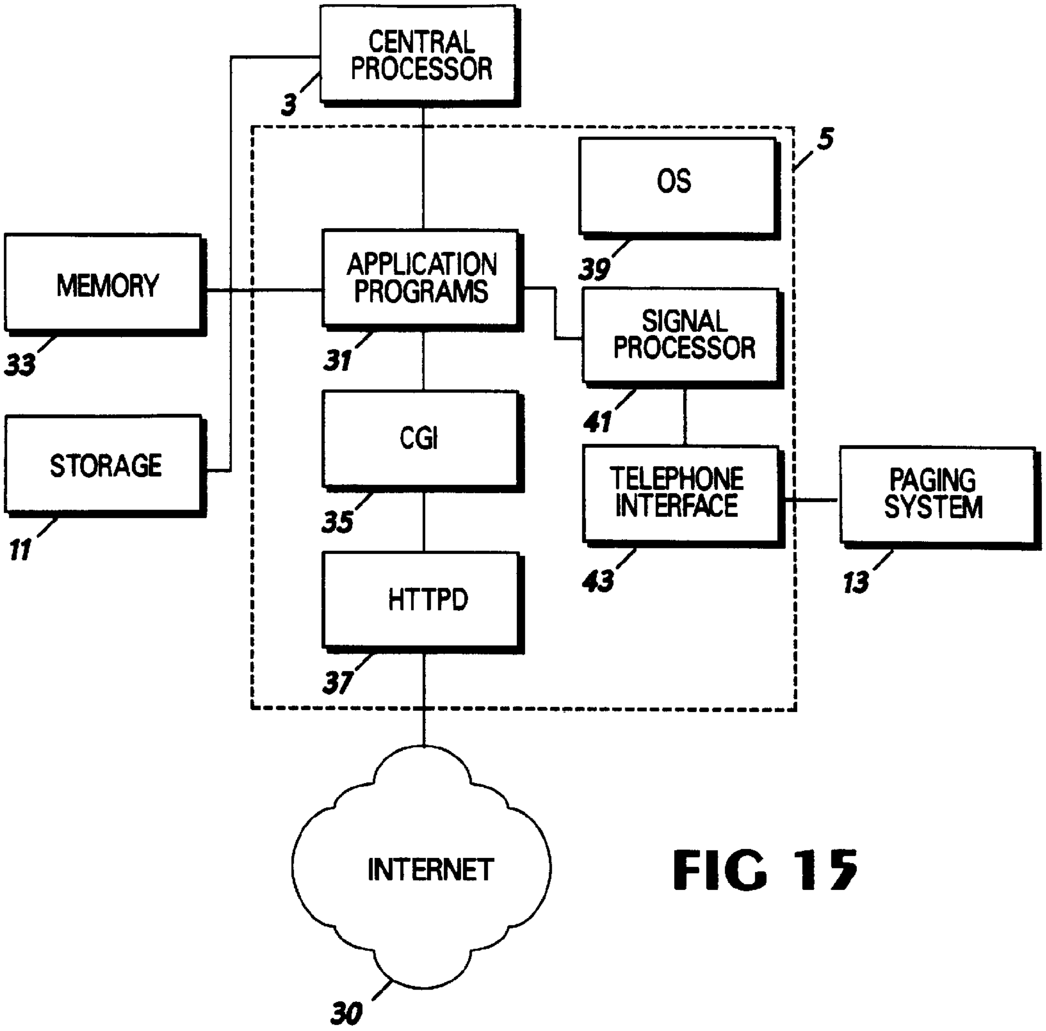


FIG 15

INDIVIDUAL APPLICATION PROGRAMS
COMMON GATEWAY INTERFACE (CGI)
HTTPD
INTERNET DEAMON (INETD)
OPERATING SYSTEM (OS)
TCP/IP

FIG 16A

PREFORMATTED HTML FILE
HTTPD
INETD
OS
TCP/IP

FIG 16B

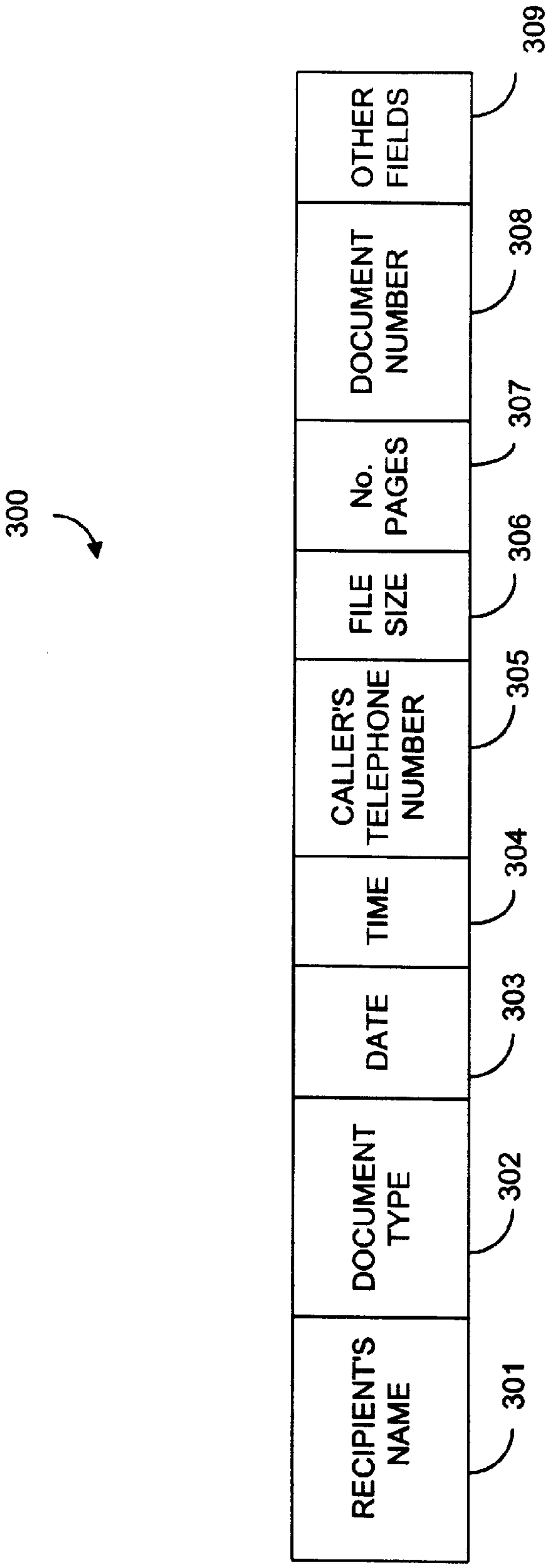

300
RECIPIENT'S NAME
301
DOCUMENT TYPE
302
DATE
303
TIME
304
CALLER'S TELEPHONE NUMBER
305
FILE SIZE
306
No. PAGES
307
DOCUMENT NUMBER
308
OTHER FIELDS
309

FIG. 17

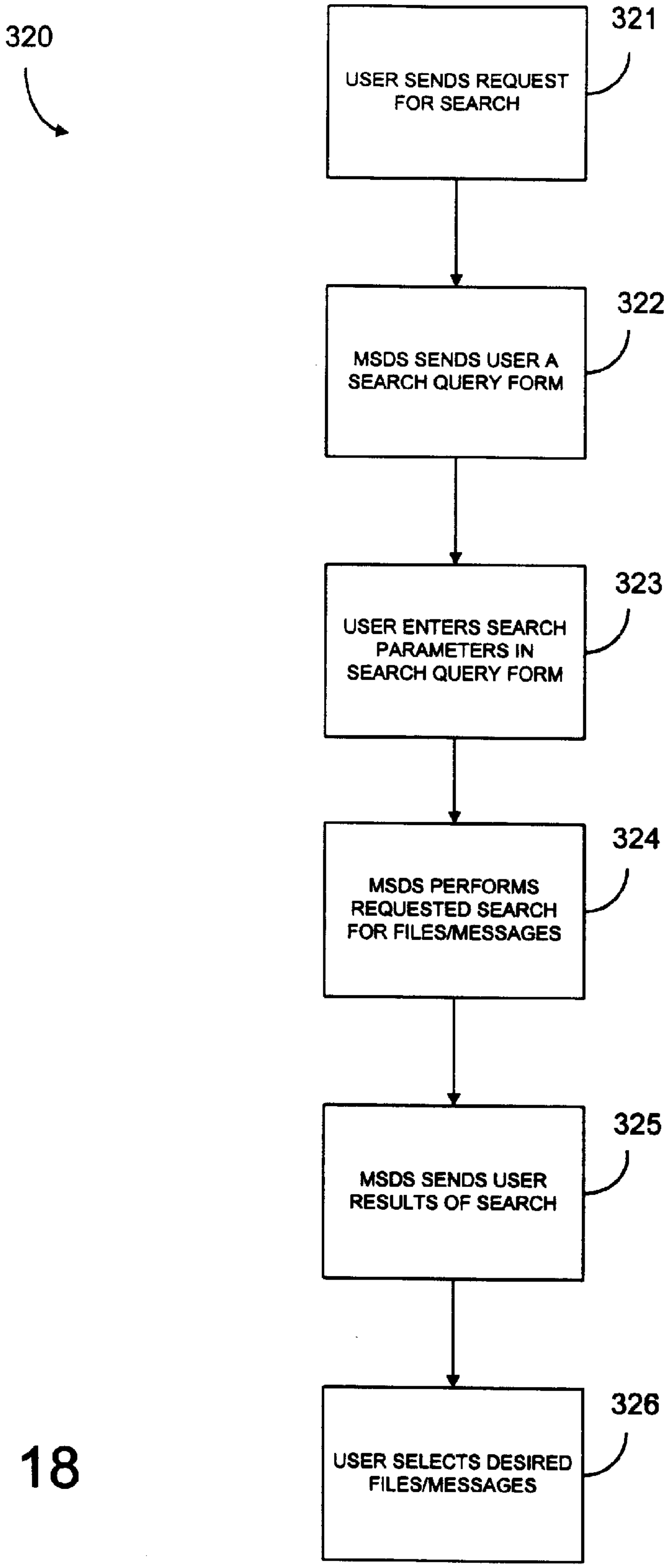
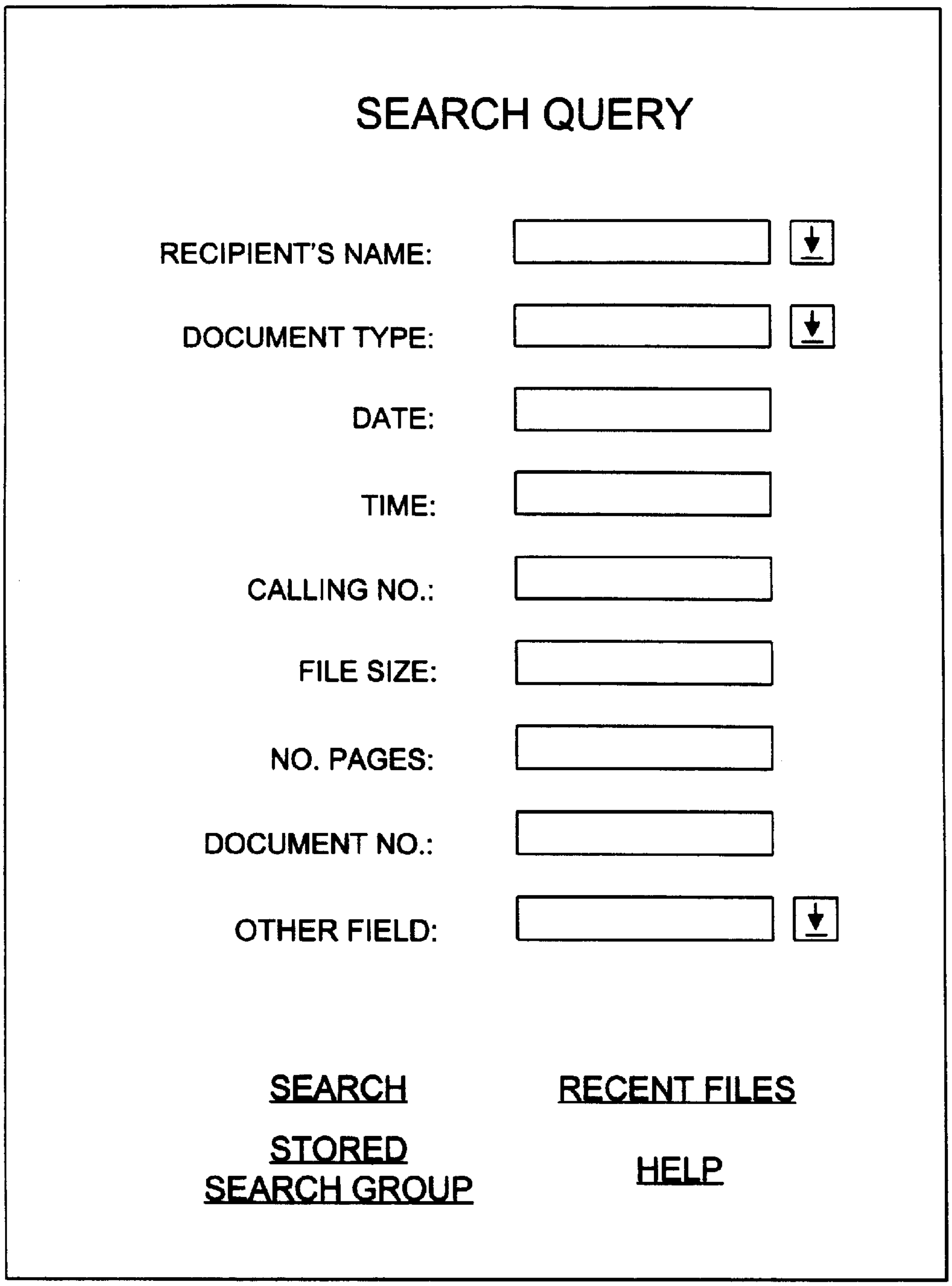

320
321
USER SENDS REQUEST FOR SEARCH
322
MSDS SENDS USER A SEARCH QUERY FORM
323
USER ENTERS SEARCH PARAMETERS IN SEARCH QUERY FORM
324
MSDS PERFORMS REQUESTED SEARCH FOR FILES/MESSAGES
325
MSDS SENDS USER RESULTS OF SEARCH
326
USER SELECTS DESIRED FILES/MESSAGES

FIG. 18



SEARCH QUERY

RECIPIENT'S NAME: ▭ ↓

DOCUMENT TYPE: ▭ ↓

DATE: ▭

TIME: ▭

CALLING NO.: ▭

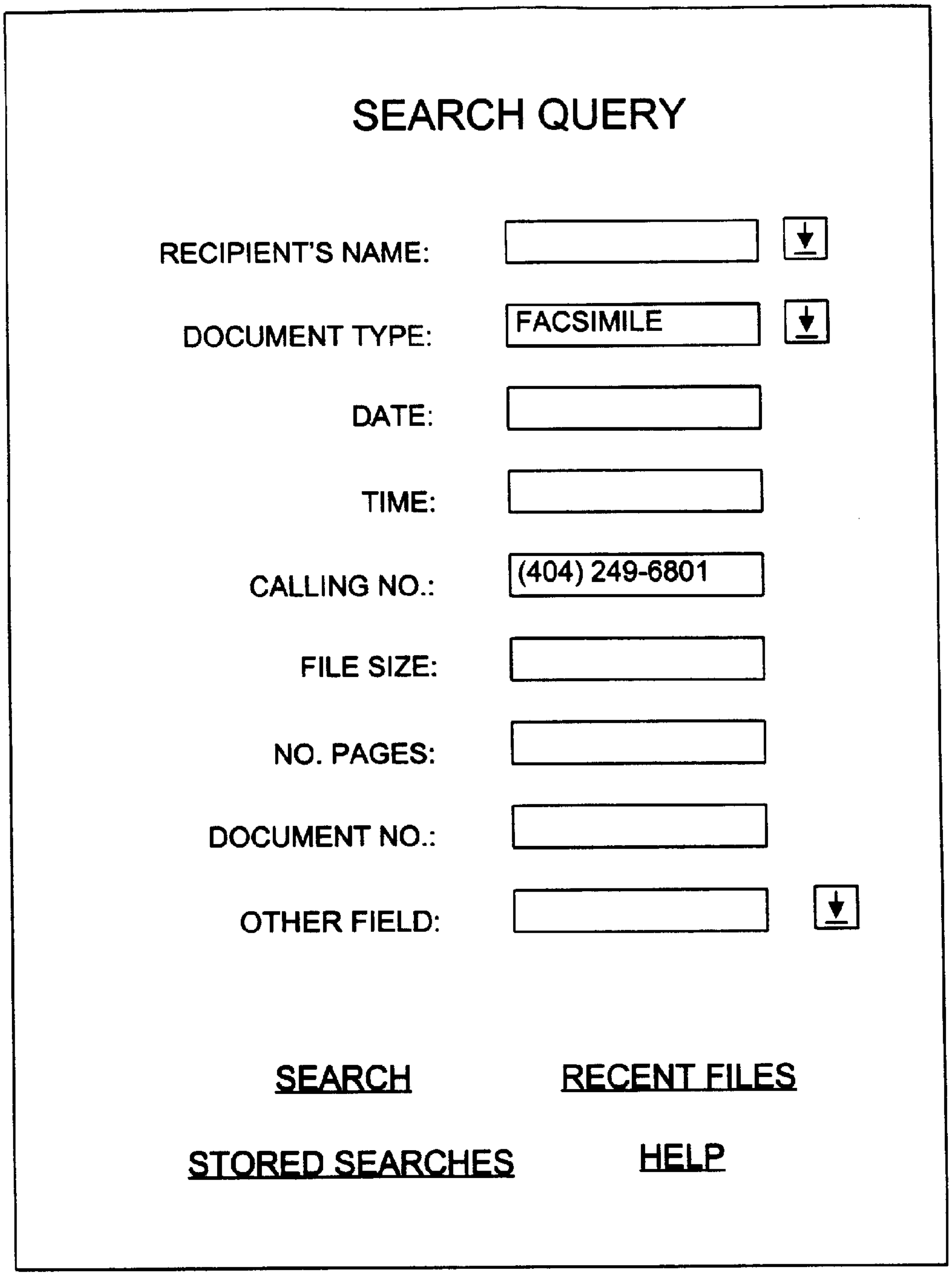
FILE SIZE: ▭

NO. PAGES: ▭

DOCUMENT NO.: ▭

OTHER FIELD: ▭ ↓

FIG. 19



SEARCH QUERY

RECIPIENT'S NAME:

DOCUMENT TYPE: FACSIMILE

DATE:

TIME:

CALLING NO.: (404) 249-6801

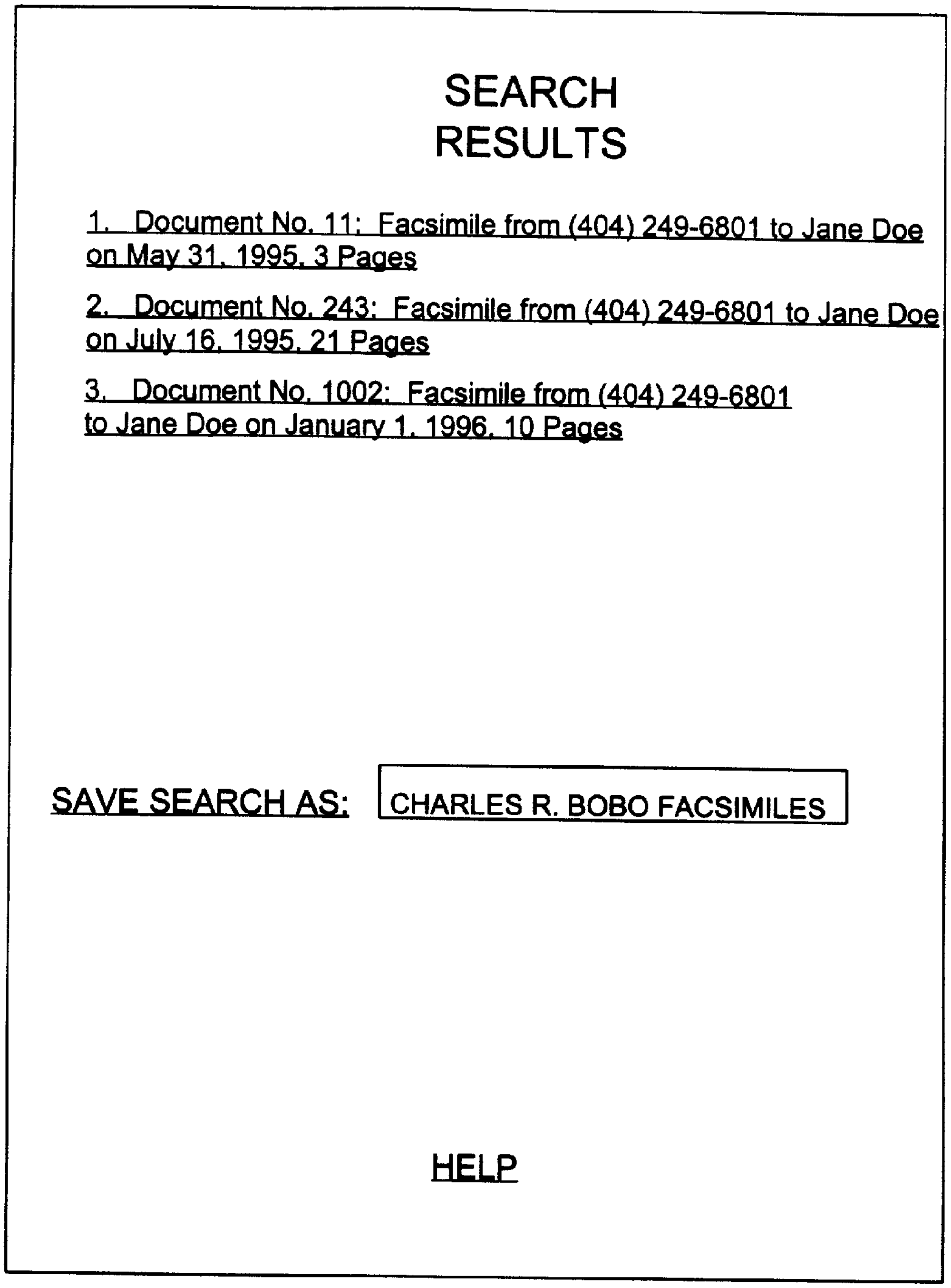
FILE SIZE:

NO. PAGES:

DOCUMENT NO.:

OTHER FIELD:

FIG. 20



SEARCH
RESULTS

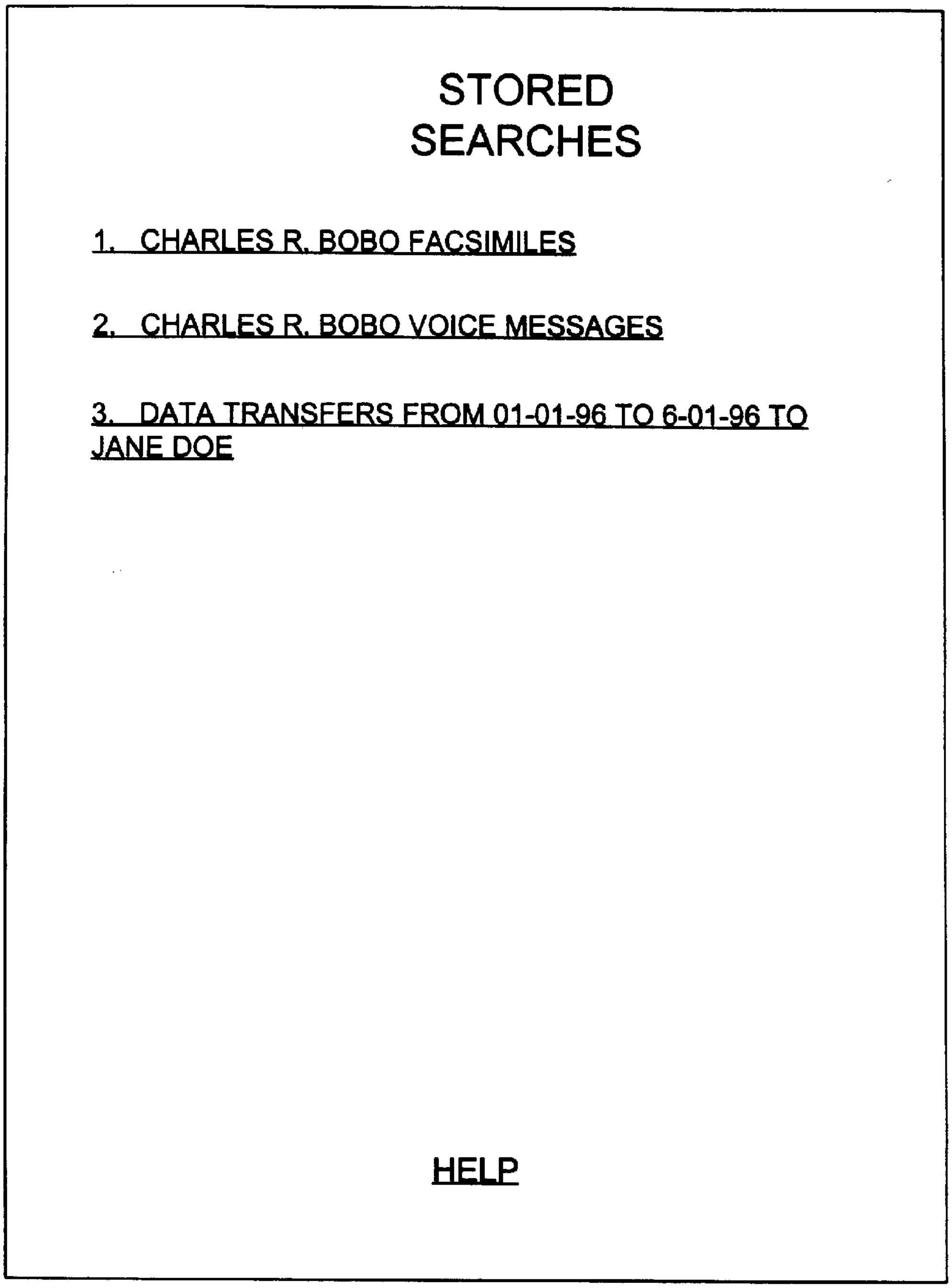
1. Document No. 11: Facsimile from (404) 249-6801 to Jane Doe on May 31, 1995, 3 Pages

2. Document No. 243: Facsimile from (404) 249-6801 to Jane Doe on July 16, 1995, 21 Pages

3. Document No. 1002: Facsimile from (404) 249-6801 to Jane Doe on January 1, 1996, 10 Pages

SAVE SEARCH AS: CHARLES R. BOBO FACSIMILES

FIG. 21



STORED
SEARCHES

1. CHARLES R. BOBO FACSIMILES

2. CHARLES R. BOBO VOICE MESSAGES

3. DATA TRANSFERS FROM 01-01-96 TO 6-01-96 TO JANE DOE

FIG. 22

SYSTEMS AND METHODS FOR STORING, DELIVERING, AND MANAGING MESSAGES

This application is a continuation-in-part of patent application Ser. No. 08/431,716, filed Apr. 28, 1995, now U.S. Pat. No. 5,675,507.

FIELD OF THE INVENTION

This invention relates to system(s) and method(s) for storing and delivering messages and, more particularly, to system(s) and method(s) for storing messages and for delivery the messages through a network, such as the Internet, or a telephone line to an intended recipient. In another aspect, the invention relates to system(s) and method(s) for storing, delivering, and managing messages or other files, such as for archival purposes or for document tracking.

BACKGROUND OF THE INVENTION

Even though the facsimile machine is heavily relied upon by businesses of all sizes and is quickly becoming a standard piece of office equipment, many businesses or households cannot receive the benefits of the facsimile machine. Unfortunately, for a small business or for a private household, a facsimile machine is a rather expensive piece of equipment. In addition to the cost of purchasing the facsimile machine, the facsimile machine also requires toner, paper, maintenance, as well as possible repairs. These expenses may be large enough to prevent many of the small businesses and certainly many households from benefiting from the service that the facsimile machine can provide. For others who are constantly traveling and who do not have an office, it may be impractical to own a facsimile machine. In fact, the Atlanta Business Chronicle estimates that 30% of the small businesses do not have any facsimile machines. Therefore, many businesses and households are at a disadvantage since they do not have access to a facsimile machine.

Because a facsimile machine can be such an asset to a company and is heavily relied upon to quickly transmit and receive documents, a problem exists in that the machines are not always available to receive a facsimile message. At times, a facsimile machine may be busy receiving another message or the machine may be transmitting a message of its own. During these times, a person must periodically attempt to send the message until communication is established with the desired facsimile machine. This inability to connect with a facsimile machine can be frustrating, can consume quite a bit of the person's time, and prevent the person from performing more productive tasks. While some more advanced facsimile machines will retry to establish communication a number of times, a person will still have to check on the facsimile machine to ensure that the message was transmitted or to re-initiate the transmission of the message.

In addition to labor costs and a reduction in office efficiency, a facsimile machine may present costs to businesses that are not readily calculated. These costs include the loss of business or the loss of goodwill that occurs when the facsimile machine is not accessible by another facsimile machine. These costs can occur for various reasons, such as when the facsimile machine is out of paper, when the machine needs repairing, or when the facsimile machine is busy with another message. These costs occur more frequently with some of the smaller businesses, who are also less able to incur these expenses, since many of them have a single phone line for a telephone handset and the facsimile machine and thereby stand to lose both telephone calls and facsimile messages when the single line is busy. In fact, the Atlanta Business Chronicle estimated that fewer than 5% of the small businesses have 2 or more facsimile machines. Many of the larger companies can reduce these losses by having more than one facsimile machine and by having calls switched to another machine when one of the machines is busy. These losses, however, cannot be completely eliminated since the machines can still experience a demand which exceeds their capabilities.

A main benefit of the facsimile machine, namely the quick transfer of documents, does not necessarily mean that the documents will quickly be routed to the intended recipient. The facsimile machines may be unattended and a received facsimile message may not be noticed until a relatively long period of time has elapsed. Further, even for those machines which are under constant supervision, the routing procedures established in an office may delay the delivery of the documents. It is therefore a problem in many offices to quickly route the facsimile message to the intended recipient.

The nature of the facsimile message also renders it difficult for the intended recipient to receive a sensitive message without having the message exposed to others in the office who can intercept and read the message. If the intended recipient is unaware that the message is being sent, other people may see the message while it is being delivered or while the message remains next to the machine. When the intended recipient is given notice that a sensitive message is being transmitted, the intended recipient must wait near the facsimile machine until the message is received. It was therefore difficult to maintain the contents of a facsimile message confidential.

In an office with a large number of employees, it may also be difficult to simply determine where the facsimile message should be routed. In light of this difficulty, some systems have been developed to automatically route facsimile messages to their intended recipient. One type of system, such as the one disclosed in U.S. Pat. No. 5,257,112 to Okada, can route an incoming call to a particular facsimile machine based upon codes entered with telephone push-buttons by the sender of the message. Another type of system, such as the one disclosed in U.S. Pat. No. 5,115,326 to Burgess et al. or in U.S. Pat. No. 5,247,591 to Baran, requires the sender to use a specially formatted cover page which is read by the system. This type of system, however, burdens the sender, who may very well be a client or customer, by requiring the sender to take special steps or additional steps to transmit a facsimile message. These systems are therefore not very effective or desirable.

Another type of routing system links a facsimile machine to a Local Area Network (LAN) in an office. For instance, in the systems disclosed in the patents to Baran and Burgess et al., after the system reads the cover sheet to determine the intended recipient of the facsimile message, the systems send an E-mail message to the recipient through the local network connecting the facsimile machine to the recipient's computer. Other office systems, such as those in U.S. Pat. No. 5,091,790 to Silverberg and U.S. Pat. No. 5,291,546 to Giler et al., are linked to the office's voice mail system and may leave a message with the intended recipient that a facsimile message has been received. Some systems which are even more advanced, such as those in U.S. Pat. No. 5,317,628 to Misholi et al. and U.S. Pat. No. 5,333,266 to Boaz et al., are connected to an office's local network and provide integrated control of voice messages, E-mail messages, and facsimile messages.

The various systems for routing facsimile messages, and possibly messages of other types received in the office, are very sophisticated and expensive systems. While these office systems are desirable in that they can effectively route the messages at the office to their intended recipients, the systems are extremely expensive and only those companies with a great number of employees can offset the costs of the system with the benefits that the system will provide to their company. Thus, for most businesses, it still remains a problem to effectively and quickly route messages to the intended recipients. It also remains a problem for most businesses to route the messages in a manner which can preserve the confidential nature of the messages.

Even for the businesses that have a message routing system and especially for those that do not have any type of system, it is usually difficult for a person to retrieve facsimile messages while away from the office. Typically, a person away on business must call into the office and be informed by someone in the office as to the facsimile messages that have been received. Consequently, the person must call into the office during normal business hours while someone is in the office and is therefore limited in the time that the information in a facsimile message can be relayed.

If the person away on business wants to look at the facsimile message, someone at the office must resend the message to a facsimile machine accessible to that person. Since this accessible machine is often a facsimile machine at another business or at a hotel where the person is lodging, it is difficult for the person to receive the facsimile message without risking disclosure of its contents. Further, since someone at the person's office must remember to send the message and since someone at the accessible facsimile machine must route the message to the person away from the office, the person may not receive all of the facsimile messages or may have to wait to receive the messages.

The retrieval of facsimile messages, as well as voice mail messages, while away from the office is not without certain costs. For one, the person often must incur long distance telephone charges when the person calls the office to check on the messages and to have someone in the office send the messages to another facsimile. The person will then incur the expenses of transmitting the message to a fax bureau or hotel desk as well as the receiving location's own charges for use of their equipment. While these charges are certainly not substantial, the charges are nonetheless expenses incurred while the person is away from the office.

Overall, while the facsimile machine is an indispensable piece of equipment for many businesses, the facsimile machine presents a number of problems or costs. Many businesses or households are disadvantaged since they are unable to reap the benefits of the facsimile machine. For the businesses that do have facsimile machines, the businesses must incur the normal costs of operating the facsimile machine in addition to the costs that may be incurred when the facsimile machine or machines are unable to receive a message. Further, the facsimile messages may not be efficiently or reliably routed to the intended recipient and may have its contents revealed during the routing process. The costs and problems in routing a facsimile message are compounded when the intended recipient is away from the office.

Many of the problems associated with facsimile messages are not unique to just facsimile messages but are also associated with voice mail messages and data messages. With regard to voice messages, many businesses do not have voice mail systems and must write the message down. Thus, the person away from the office must call in during normal office hours to discover who has called. The information in these messages are usually limited to just the person who called, their number, and perhaps some indication as to the nature of the call. For those businesses that have voice mail, the person away from the office must call in and frequently incur long distance charges. Thus, there is a need for a system for storing and delivery voice messages which can be easily and inexpensively accessed at any time.

With regard to data messages, the transmission of the message often requires some coordination between the sender and the recipient. For instance, the recipient's computer must be turned on to receive the message, which usually occurs only when someone is present during normal office hours. Consequently, the recipient's computer is usually only able to receive a data message during normal office hours. Many households and also businesses may not have a dedicated data line and must switch the line between the phone, computer, and facsimile. In such a situation, the sender must call and inform the recipient to switch the line over to the computer and might have to wait until the sender can receive the message. The retransmission of the data message to another location, such as when someone is away from the office, only further complicates the delivery. It is therefore frequently difficult to transmit and receive data messages and is also difficult to later relay the messages to another location.

A standard business practice of many companies is to maintain records of all correspondence between itself and other entities. Traditionally, the correspondence that has been tracked and recorded includes letters or other such printed materials that is mailed to or from a company to the other entity. Although tracking correspondence of printed materials is relatively easy, non-traditional correspondence, such as facsimile messages, e-mail messages, voice messages, or data messages, are more difficult to track and record.

For example, facsimile messages may be difficult to track and record since the messages may be received on thermal paper, which suffers from a disadvantage that the printing fades over time. Also, accurate tracking of facsimile messages is difficult since the facsimile messages may only be partially printed at the facsimile machine or the messages may be lost or only partially delivered to their intended recipients. Facsimile messages also present difficulties since they are often delivered within an organization through different channels than ordinary mail and thus easily fall outside the normal record keeping procedures of the company.

Voice mail messages are also difficult to track and record. Although voice messages can be saved, many voice mail servers automatically delete the messages after a certain period of time. To maintain a permanent record of a voice message, the voice message may be transcribed and a printed copy of the message may be kept in the records. This transcribed copy of the voice message, however, is less credible and thus less desirable than the original voice message since the transcribed copy may have altered material or may omit certain portions of the message.

In addition to facsimile and voice mail messages, data messages are also difficult to track and record. A download or upload of a file may only be evident by the existence of a file itself. A file transfer procedure normally does not lend itself to any permanent record of what file was transferred, the dialed telephone number, the telephone number of the computer receiving the file, the time, or the date of the transfer. It is therefore difficult to maintain accurate records of all data transfers between itself and another entity.

SUMMARY OF THE INVENTION

It is an object of the invention to reliably and efficiently route messages to an intended recipient.

It is another object of the invention to route messages to the intended recipient while maintaining the contents of the message confidential.

It is another object of the invention to enable the intended recipient to access the messages easily and with minimal costs.

It is a further object of the invention to permit the simultaneous receipt of more than one message on behalf of the intended recipient.

It is a further object of the invention to enable the intended recipient of a message to access the message at any time and at virtually any location world-wide.

It is yet a further object of the invention to enable the intended recipient of a message to browse through the received messages.

It is yet a further object of the invention to quickly notify an intended recipient that a message has been received.

It is still another object of the invention to receive messages of various types.

It is still another object of the invention to deliver messages according to the preferences of the intended recipient.

It is still a further object of the invention to record and track correspondence, such as facsimile messages, voice mail messages, and data transfers.

Additional objects, advantages and novel features of the invention will be set forth in the description which follows, and will become apparent to those skilled in the art upon reading this description or practicing the invention. The objects and advantages of the invention may be realized and attained by the appended claims.

To achieve the foregoing and other objects, in accordance with the present invention, as embodied and broadly described herein, a system and method for storing and delivering messages involves receiving an incoming call and detecting an address signal associated with the incoming call, the address signal being associated with a user of the message storage and delivery system. A message accompanied with the address signal is then received and converted from a first file format to a second file format. The message is stored in the second file format within a storage area and is retrieved after a request has been received from the user. At least a portion of the message is then transmitted to the user over a network with the second file format being a mixed media page layout language.

In another aspect, a network message storage and delivery system comprises a central processor for receiving an incoming call, for detecting an address signal on the incoming call, for detecting a message on the incoming call, and for placing the message in a storage area. The address signal on the incoming call is associated with a user of the network message storage and delivery system. A network server receives the message from the storage area, converts the message into a mixed media page layout language, and places the message in the storage area. When the network server receives a request from the user over the network, the network server transmits at least a portion of the message over the network to the user.

Preferably, the network storage and delivery system can receive facsimile messages, data messages, or voice messages and the network is the Internet. The messages are converted into a standard generalized mark-up language and

the user is notified that a message has arrived through E-mail or through a paging system. A listing of the facsimile messages may be sent to the user in one of several formats. These formats include a textual only listing or a listing along with a full or reduced size image of the first page of each message. A full or reduced size image of each page of a message in the listing may alternatively be presented to the user.

According to a further aspect, the invention relates to a system and method for managing files or messages and involves storing message signals in storage and receiving requests from a user for a search. The search preferably comprises a search query that is completed by a user and supplied to a hyper-text transfer protocol deamon (HTTPD) in the system. The HTTPD transfers the request through a common gateway interface (CGI) to an application program which conducts the search. The results of the search are preferably returned through the HTTPD to the computer in the form of a listing of all messages or files satisfying the search parameters. The user may then select one or more of the listed messages or files and may save the search for later references.

BRIEF DESCRIPTION OF THE DRAWINGS

The accompanying drawings, which are incorporated in, and form a part of, the specification, illustrate an embodiment of the present invention and, together with the description, serve to explain the principles of the invention. In the drawings:

FIG. **1** is a block diagram illustrating the connections of a message storage and delivery system MSDS;

FIG. **2** is an overall flow chart of operations for transmitting a message to the MSDS of FIG. **1**;

FIG. **3** is an overall flow chart of operations for receiving a message stored at the MSDS of FIG. **1**;

FIGS. **4**(A) and (B) are flowcharts of operations for generating HTML files according to user preferences;

FIG. **5** is a flowchart of operations for generating requested information;

FIG. **6** is a flowchart of operations for converting a facsimile message into HTML files;

FIG. **7** is an exemplary display of a first page of a facsimile message according to a fourth display option;

FIG. **8** is a flowchart of operations for converting a voice message into an HTML file;

FIG. **9** is a flowchart of operations for converting a data message into an HTML file;

FIG. **10** is a flowchart of operations for detecting a type of call received at the MSDS **10**;

FIG. **11** is a flowchart of operations for receiving voice messages;

FIG. **12** is a flowchart of operations for interacting with an owner's call;

FIG. **13** is a more detailed block diagram of the MSDS **10**;

FIG. **14** is a block diagram of the central processor in FIG. **13**;

FIG. **15** is a block diagram of the Internet Server of FIG. **13**;

FIGS. **16**(A) and **16**(B) depict possible software layers for the Internet Server of FIG. **13**;

FIG. **17** is a diagram of a data entry for a message signal;

FIG. **18** is a flowchart of a process for sending a search query, for conducting a search, and for returning results of the search to a computer through the Internet;

FIG. 19 is an example of a search query form for defining a desired search;

FIG. 20 is an example of a completed search query;

FIG. 21 is an example of a set of search results returned to the computer in response to the search query of FIG. 20; and

FIG. 22 is an example of a listing of stored searches.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

Reference will now be made in detail to the preferred embodiments of the invention, examples of which are illustrated in the accompanying drawings.

With reference to FIG. **1**, a message storage and delivery system (MSDS) **10** is connected to a central office **20** of the telephone company through at least one direct inward dialing (DID) trunk **15**. With each call on the DID trunk **15**, an address signal indicating the telephone number being called is provided to the MSDS **10**. The DID trunk **15** can carry a large number of telephone numbers or addresses. Preferably, the DID trunk **15** comprises a number of DID trunks **15** connected in parallel between the central office **20** and the MSDS **10** so that the MSDS **10** can simultaneously receive more than one call and, moreover, can simultaneously receive more than one call for a single telephone number or address.

The central office **20** is connected to a number of third parties. For instance, the central office **20** may be connected to a facsimile machine **24**, a telephone set **26**, and to a computer **28** with each connection being made through a separate telephone line. While a single computer **28** is shown in the figure, the single computer **28** may actually represent a local area network which is connected through the central office **20** to the MSDS **10**. Although the facsimile machine **24**, telephone set **26**, and computer **28** have been shown on separate lines, it should be understood that one or more of these devices could share a single line.

The MSDS **10** is also connected to a network, preferably the Internet World Wide Web **30**. Although the Internet **30** has been shown as a single entity, it should be understood that the Internet **30** is actually a conglomeration of computer networks and is a constantly evolving and changing structure. The MSDS **10** therefore is not limited to the current structure or form of the Internet **30** but encompasses any future changes or additions to the Internet **30**. Further, the MSDS **10** is shown as being directly connected to the Internet **30**, such as through its own node or portal. The MSDS **10**, however, may be practiced with any suitable connection to the Internet **30**, such as through an intermediate Internet access provider.

With reference to FIG. 2 depicting an overall operation of the invention, a telephone call directed to a number serviced by the MSDS **10** is initiated at step **40** by a third party, for instance, through the facsimile machine **24**, telephone set **26**, or computer **28**. The incoming telephone call may therefore carry a facsimile message, a voice message, or a data message. At step **42**, the address signal associated with the initiated call is routed through the central office **20**, over the DID trunk **15**, and to the MSDS **10**.

When the call reaches the MSDS **10**, the call is routed within the MSDS **10** in a manner that will be described in more detail below with reference to FIG. **13**. At step **46**, the MSDS **10** answers the telephone call and receives the address signal from the DID trunk **15**. Next, at step **48**, the call is established between the MSDS **10** and the third party and, at step **50**, the MSDS **10** receives the message transmitted over the telephone line. The message is stored at step **52**, a database within the MSDS **10** is updated at step **54**, and the intended recipient of the message is notified at step **56**. The intended recipient of the message uses the services provided by the MSDS **10** and will hereinafter be referred to as a user. At step **58**, the message is converted into hyper-text mark-up language (HTML).

After the MSDS **10** receives a message for one of its users, the user can then communicate with the MSDS **10** at any time and at any location by connecting to the Internet World Wide Web **30** and retrieving the message stored within the MSDS **10**. With reference to FIG. **3**, at step **60** the user first connects to the Internet **30**, such as through a personal computer **32** which may be connected to the Internet **30** in any suitable manner, such as through its own portal or node or through some intermediate access provider. The personal computer **32** is not limited to a single computer but may instead comprise a network of computers, such as a local area network within an office.

Once connected with the Internet **30**, at step **62**, the user accesses with a hyper-text browser the Universal Resource Locator (URL) associated with his or her MSDS **10** mailbox. The computer **32** may use any suitable hypertext browser, such as Netscape, to access the mailbox. A Hypertext Transfer Protocol Deamon (HTTPD) within the MSDS **10** receives the URL request at step **64** and, at step **66**, requests user authentication. The user then supplies his or her ID and password at step **68** and, if found valid at step **70**, the MSDS **10** provides the computer **32** with access to the mailbox at step **72**. If the ID and password are invalid, as determined at step **70**, then the HTTPD sends the computer **32** an authentication failure message at step **74**.

After the user gains access to the mailbox at step **72**, the user can request information stored within the MSDS **10**. The MSDS **10** receives the request at step **76** and, at step **78**, determines whether the information exists. As is common practice, the MSDS **10** also determines the validity of the request at step **78**. The request from the user will include the mailbox number for the user, the message identifier, display preferences, and, if the message is a facsimile message, a page identifier. If for any reason the request is invalid, such as when a hacker is attempting to gain access to privileged information, the request for the information will be terminated.

If the requested information is available, then at step **80** the information is transmitted through the Internet **30** to the user's computer **32**. If, on the other hand, the information does not exist, then at step **82** the MSDS **10** will generate the requested information and then send the information to the user's computer through the Internet **30** at step **80**.

Prior to gaining access to the mailbox at step **72**, the user is preferably sent a greeting page or other such type of information which permits the user to learn about the services provided by the MSDS **10**, open an account with the MSDS **10**, or gain access to an account. Once access is provided at step **72**, the user is provided with information indicating the total number of messages stored in his or her mailbox within the MSDS **10**. Preferably, the information sent by the MSDS **10** indicates the total number of messages for each type of message and also the total number of saved messages versus the total number of new messages.

The user is also preferably given the option at this step to change account information. The account information might include the E-mail address for the user, the manner in which messages are to be reviewed, the user's pager information, as well as other user preferences. The display options and other user preferences will be discussed in further detail below.

The general information HTML file which indicates the total number of different messages is provided with a number of anchors, which are also termed links or references. In general, an anchor permits a user on the computer **32** to retrieve information located on another file. For instance, an anchor to a listing of facsimile messages is preferably provided on the display of the total number of messages. When the user selects the anchor for the facsimile list, the MSDS **10** pulls up and displays the file containing the list of facsimiles, such as a file "faxlist.html. " The other types of messages, such as voice messages and data messages, would have similar anchors on the general information page directed to their respective HTML listing files.

When a new message is received at step **54** in FIG. **2**, the user's mailbox is updated to display the total number and types of messages. The MSDS **10** might also update other files in addition to the total listing of messages. Additionally, at this time, the MSDS **10** sends an E-mail message to the user's computer **32** to inform the user of the newly arrived message. The MSDS **10** could also send notice to the user through a paging system so that the user receives almost instantaneous notice that a message is received.

The MSDS **10** also generates additional information according to the user's preferences. These preferences on how the MSDS **10** is configured for the user include options on how the messages are reviewed. With facsimile messages, for instance, the user can vary the amount or the type of information that will be supplied with the listing of the facsimile messages by selecting an appropriate option. Other options are also available so that the user can custom fit the MSDS **10** to the user's own computer **32** or own personal preferences.

For instance, when a facsimile message is received, the MSDS **10**, at step **54**, will update the total listing of all messages to indicate the newly received message and may additionally generate the HTML files for the newly received facsimile message according to the user's preferences. When the user later requests information on the message at step **76**, the HTML information has already been generated and the MSDS **10** may directly send the requested information to the user at step **80**. If, on the other hand, the user desires to view the message according to one of the other options, the MSDS **10** will generate the HTML files at step **82** according to that other option at the time of the request.

A first option available to the user for viewing a facsimile message is a textual only listing of the messages. The information on the textual listing preferably includes the date and time that the message was received at the MSDS **10**, the telephone number from where the message was transmitted, the number of pages, the page size, and the size of the message in bytes. The messages, of course, could be listed with other types of information. When the user selects one of the facsimile messages on the list, a request is sent to the HTTPD within the MSDS **10** causing the message to be downloaded via the Internet **30** to the user's computer **32**. Once the message is received by the computer **32**, the message can be displayed, printed, or saved for further review.

The second through fifth options allow the user to preview an image of the facsimile message before having the message downloaded from the MSDS **10** through the Internet **30** and to the computer **32**. The second option permits the user to view the list of messages with a reduced size image of the cover page next to each entry on the list. When the user selects one of the messages on the list, the selected facsimile message is transmitted through the Internet **30** to the computer **32**. The user may also scroll through the listings if all of the message cannot be displayed at one time on the computer **32**.

The third option provides the user with a full size view of the cover page of each facsimile message. The user can quickly scroll through the cover pages of each message without downloading the entire message to the computer **32**. The full size view of the cover pages permit the user to clearly discern any comments that may be placed on the cover page, which may not be possible from just a reduced image of the cover page available through the second option.

The fourth option provides the user with a reduced size image of each page and permits the user to scroll through the entire message. The user can therefore read the entire facsimile message on screen before the message is downloaded onto the computer **32**. With this option, the user can go through the pages of the facsimile message and can also skip to the next message or previous message. Additionally, the user has the option of enlarging a page to a full size view of the page. When one of the messages is selected, as with the other options, the HTTPD within the MSDS **10** causes the facsimile message to be transmitted through the Internet **30** to the user's computer **32**.

With a fifth option, a full size image of each page is transmitted to the user's computer **32**. The user can scroll through the pages of the facsimile message and easily read the contents of each page. If the user wants the message downloaded to the computer **32**, the user selects the message and the HTTPD within the MSDS **10** transmits the message to the user's computer **32** through the Internet **30**.

As discussed above, after the database is updated at step **54**, the MSDS **10** will generate additional information based upon the option selected for displaying the facsimile messages. More specifically, as shown in FIG. **4**(A), if the first option has been selected, as determined at step **100**, then at step **102** the MSDS **10** will generate the textual listing of the facsimile messages with anchors or references to the respective facsimile files. The HTML files are then moved to an Internet Server at step **104**.

If the first option is not selected, the MSDS **10** next determines whether the second option has been selected at step **106**. With the second option, the facsimile messages are listed along with a reduced size image of the cover page. To generate this information, the cover page is extracted from the facsimile file at step **108** and a reduced size HTML image of the cover page is created at step **110**. At step **112**, a listing of the facsimile messages is generated with a thumbnail view of each cover page linked to its respective facsimile file. The generated HTML files are then sent to the Internet Server at step **104**.

When the third option is selected, as determined at step **114**, a full size image of the cover page is sent to the computer **32**. The full size image of the cover page is generated by first extracting the cover page from the facsimile file at step **116**. Next, the cover page is converted into a full size HTML image at step **118** and, at step **120**, the listing is generated with the embedded cover page linked to the facsimile file.

If, at step **122**, the fourth option is determined to be selected, then a reduced size image of each page is provided to the user with the option of enlarging the page to view the contents of the page more clearly. With reference to FIG. 4(B), the information necessary for the third option is produced by first extracting the first page of the facsimile message at step **124**. A reduced size HTML image is created at step **126** and then a full size HTML image is created at step **128**. At step **130**, the listing is generated with embedded thumbnail images of the pages with links to the full size images. If the page is not the last page, as determined at step **140**, then the next page is extracted at step **142** and steps **126** to **130** are repeated to generate the HTML files for the other pages of the facsimile message. After the last page has been converted into an HTML file according to the third option, the files are moved onto the Internet Server at step **104**.

At step **144**, the MSDS **10** determines whether the fifth option has been selected. The fifth option provides the user with a full size image of each page of the facsimile message. While only five options have been discussed, the invention may be practiced with additional options. Consequently, with additional options and with the fourth option not being selected, the MSDS **10** would next determine whether one of the additional options have been selected. With the preferred embodiment of the invention having only five options, however, the MSDS **10** will assume that the fifth option has been selected if none of the first four options were found to be selected.

The information necessary to display the pages of the facsimile message according to the fifth option is generated by first extracting the first page of the facsimile message at step **146**. At step **148**, a full size HTML image of the page is created and, at step **150**, a listing is generated with an embedded image and links to previous and next pages. When the page is not the last page, as determined at step **152**, the MSDS **10** extracts the next page and generates the HTML file for that page. After all pages have been converted into HTML files according to the fourth option, the files are sent to the Internet Server at step **104**.

While FIGS. **4**(A) and (B) describe the operations of the MSDS **10** at the time a message is received, FIG. **5** depicts an overall flowchart of operations for the MSDS **10** when the user requests a page of information in a display format other than the user's preferred option of displaying the message. FIG. **5** is therefore a more detailed explanation of how the MSDS **10** generates the necessary information at step **82** of FIG. **3**.

In general, as shown in FIG. **5**, the MSDS **10** first determines the type of image that is needed at step **82***a*. For example, at this step, the MSDS **10** will determine whether images are unnecessary, whether an image of just the cover page is necessary, whether an image is needed for every page, and whether the image needs to be a full size, a reduced size, or both full and reduced sized images. At step **82***b*, the MSDS **10** determines whether the image has already been created. If the image has not been created, then at step **82***c* the MSDS **10** will extract the page from the base facsimile file and, at step **82***d*, generate the required HTML image. As discussed above, the required image may be for just the cover page, for all the pages, and may be a full size and/or a reduced size image of the page. At step **82***e*, the image is embedded with links or anchors to other HTML files. These links or anchors might be references to the next and previous pages and also to the next and previous facsimile messages. Finally, the HTML file having the embedded image and links is sent to the user at step **80** in FIG. **3**.

The process for converting a facsimile message into HTML files according to the fifth option will be described with reference to FIG. **6**. This process will occur at step **54** when the message is received and when the fifth option is the user's preferred option of displaying the messages. It should be understood that a similar type of process will also occur when the user requests a page of information according to the fifth option when the user is retrieving a facsimile message and the fifth option is not the user's preferred option. The conversion processes according to the other options will become apparent to those skilled in the art and will therefore not be discussed in further detail.

With reference to FIG. **6**, when the facsimile message is received, the message is in a Tagged Image File Format/Facsimile (TIFF/F) and each page of the facsimile message is split into a separate file. Each page of the facsimile message is then converted from the TIFF/F format into a Portable Pixel Map (PPM) format. The PPM files are next converted into separate Graphic Interchange Format (GIF) files and then into separate HTML files. Thus, each page of the facsimile message is converted into a separate HTML file. The TIFF/F files may be converted into PPM with an available software package entitled "LIBTIFF" and the PPM files may be converted into GIF files with an available software package found in "Portable Pixel Map Tools."

The invention is not limited to this exact conversion process or to the particular software packages used in the conversion process. For instance, the TIFF/F files may be converted into another portable file format, through any other type of intermediate format, or may be converted directly into the GIF format. Further, instead of GIF, the facsimile messages may be converted into JPEG, BMP, PCX, PIF, PNG, or any other suitable type of file format.

The files may be identified with any suitable filename. In the preferred embodiment, the files for each user are stored in a separate directory assigned to just that one user because an entire directory for a given user generally can be protected easier than the individual files. The memory, however, may be organized in other ways with the files for a single user being stored in different directories. The the first part of the filename is a number preferably sequentially determined according to the order in which messages arrive for that user. The preferred naming convention for ending the filenames is depicted in FIG. **6**. Each page of the facsimile message is saved as a separate file with an extension defined by the format of the file. Thus, the files will end with an extension of ".TIFF," ".PPM," ".GIF," or ".HTML" according to the format of the particular file. In the example shown, the separate pages have filenames which end with the respective page number, for instance, the first page ends with a "1." The files, however, are preferably terminated with a letter or multiples letters to indicate the order of the pages. For instance, page 1 might have an ending of "aa," page 2 might have an ending of "ab," etc. The invention, however, is not limited to the disclosed naming convention but encompasses other conventions that will be apparent to those skilled in the art.

As shown in FIG. **6**, in addition to the GIF files representing the pages of the facsimile message, the HTML files include a number of anchors or references. In the example shown, the first HTML file has an anchor a for the "Next Page." Anchor a is defined as a=<A HREF="2.html">Next Page</a> and will therefore reference the second HTML file when a user selects the "Next Page." The second HTML file has an anchor b for the "Previous Page" and an anchor c for the "Next Page" and the third HTML file has an anchor d for the "Previous Page." With these particular HTML files, the user can scroll through each page of the facsimile message and view a full size image of the page.

Each HTML file preferably contains anchors in addition to those relating to "Next Page" and "Previous Page." For instance, each HTML file may contain an anchor to the next facsimile message, an anchor to the previous facsimile message, and an anchor to return to the facsimile list. The HTML files preferably contain anchors relating to “Save” and “Delete.” When the “Save” anchor is selected, the user would be able to save the message under a more descriptive name for the message. The “Delete” anchor is preferably followed by a inquiry as to whether the user is certain that he or she wants to delete the message. Other anchors, such as an anchor to the general listing, will be apparent to those skilled in the art and may also be provided.

FIG. **7** provides an example of a display according to the fifth option for the first page of the facsimile message shown in FIG. **6**. The headings of the display provide information on the telephone number from where the message was sent, the date and time the message was received at the MSDS **10**, and an indication of the page of the message being displayed. The main portion of the display is the full size image of the page. At the bottom of the display, an anchor or link is provided to the “Next Page” and another anchor is provided to the “Return to Fax Listing.” Additional information may also be provided on the display, such as a link to a company operating the MSDS **10**.

An example of the “1.html” file for generating the display shown in FIG. **7** is shown below in Table 1.

```
<HTML>
<HEAD>
<TITLE>Fax Received on May 31, 1995 at 1:58 PM from
  (404)249 6801; Page 1 of 3</TITLE>
</HEAD>
<BODY>
<H1>Fax from (404)249-6801</H1>
<H2>Received on May 31, 1995 at 1:58 PM</H2>
<H2>Page 1 of3</H2>
<IMG SRC=“1.gif”>
<P>
<A HREF=“2.html”>Next Page</a>
<HR>
<A HREF=“faxlist.html”>Return to Fax Listing</A>
<P>
This page was automatically generated by FaxWeb(tm) on
May 31, 1995 at 2:05 PM.
<P>
&copy; 1995 NetOffice, Inc.
<HR>
<Address>
<A HREF=“http://www.netoffice.com/”>NetOffice,
  Inc.</A><BR>
PO Box 7115<BR>
Atlanta, Ga. 30357<BR>
<A HREF=
  “mailto:info@netoffice.com”>info@netoffice.com</A>
</Address>
</BODY>
</HTML>
```

TABLE 1

As is apparent from the listing in Table 1, the image file “1.gif” for the first page is embedded into the HTML file “1.html.” Also apparent from the listing is that the anchor for “Next Page” directs the MSDS **10** to the second page of the facsimile message having the filename “2.html” and the anchor for “Return to Fax Listing” directs the MSDS **10** to the filename “faxlist.html” containing the list of facsimile messages.

A process for converting a voice message into an HTML file is illustrated in FIG. **8**. The voice message is originally stored in a VOX format or an AD/PCM format and is retrieved at step **170**. The voice message is then converted either into an AU format or WAV format in accordance with the user's preference, which is stored in memory. Preferably, the message is preferably in the AD/PCM format originally and is converted in WAV, but the voice files may alternatively be stored and converted in file formats other than the ones disclosed, such as RealAudio (RA).

At step **174**, the listing of all of the voice messages is then updated to include a listing of the newly received voice message and an anchor to the voice message. For instance, the original voice message may be stored with filename “1.vox” and is converted into WAV and stored with a filename “1.wav.” The HTML file “voicelist.html” which contains a list of all voice messages would then have an anchor to the filename “1.wav” along with identifying information for the voice message, such as when the message was received.

The listing of the voice messages may have additional anchors or references. For instance, each voice message may have an anchor directing the MSDS **10** to a file which contains a short sampling of the message. Thus, when the user selects this anchor, the user could receive the first 5 seconds of the message or some other predefined number of seconds. As with the listing of facsimile messages, the listing of the voice messages also preferably has anchors to “Save” and “Delete.”

FIG. **9** illustrates a process for converting a data message into HTML. At step **180**, the data file is retrieved from a database and at step **182** the HTML file containing the list of data messages is updated to include a listing of the newly received message along with identifying information. For instance, the HTML file for the listing “datalist.html” would be updated to include an anchor to a data file “file 1.1” and would have information such as the time and date that the data was transmitted, the size of the data file, as well as additional identifying information.

Because the MSDS **10** can receive messages of various types, such as a facsimile message, voice message or data message, the MSDS **10** must be able to determine the type of message that is being sent over the DID trunk **15**. With reference to FIG. **10**, when an incoming call is received, the MSDS **10** goes off hook at step **200** and starts to generate a ringing sound. If, at step **202**, a facsimile calling tone is detected, then the ringing sound is stopped at step **204** and the message is received as a facsimile message at step **206**. Similarly, when a data modem calling tone is detected at step **208**, the ringing sound is stopped at step **210** and the message is identified as a data message at step **212**.

If the MSDS **10** detects a DTMF digit at step **214**, the ringing sound is stopped at step **216** and the MSDS **10** then determines which digit was pressed. When the digit is a “1,” as determined at step **218**, the message is identified as a facsimile message. The MSDS **10** will thereafter receive and store the facsimile message in the manner described above with reference to FIG. **2**. If the digit is identified as a “0” at step **220**, the call is identified as an owner's call and will be processed in a manner that will be described below with reference to FIG. **12**. As will be apparent, other digits may cause the MSDS **10** to take additional steps. If any other DTMF digit is pressed, at step **224** the MSDS **10** activates a voice call system, which will be described in more detail below with reference to FIG. **11**.

With step **226**, the MSDS **10** will enter a loop continuously checking for a facsimile calling tone, a data modem calling tone, or for a DTMF digit. If after n rings none of these tones or digits has been detected, the ringing sound is stopped at step **228** and the voice call system is activated at step **224**.

With reference to FIG. **11**, when a fax calling tone or modem calling tone is not detected, the voice call system begins at step **230** by playing a voice greeting. If the greeting was not interrupted by a DTMF digit as determined at step **232**, then the caller is prompted for the voice message at step **234** and, at step **236**, the voice message is recorded and stored in memory. At step **238**, the caller is prompted with a number of options, such as listening to the message, saving the message, or re-recording the message. Since the selection of these options with DTMF digits will be apparent to those skilled in the art, the details of this subroutine or subroutines will not be described in further detail. When the caller wishes to re-record the message, as determined at step **240**, the caller is again prompted for a message at step **234**. If the caller does not wish to re-record the message, the call is terminated at step **242**.

If the voice greeting is interrupted by a DTMF digit, as determined at step **232**, then the MSDS **10** ascertains which digit has been pressed. At step **244**, if the digit is a "0," the MSDS **10** detects that the call is an owner's call. When the digit is a "1," the MSDS **10** is informed at step **206** that the call carries a facsimile message. As discussed above with reference to FIG. **10**, other DTMF digits may cause the MSDS **10** to take additional steps. If an invalid digit is pressed, by default at step **248** the routine returns to step **234** of prompting the caller for a message.

It should be understood that the invention is not limited to the specific interactive voice response system described with reference to FIG. **11**. As discussed above, the invention may be responsive to DTMF digits other than just a "0" and a "1." Further variations or alterations will be apparent to those skilled in the art.

With reference to FIG. **12**, when the call is considered an owner's call, the caller is first prompted for the password at step **250**. The password is received at step **252** and, if found correct at step **254**, a set of announcements are played to the owner. These announcements would preferably inform the owner of the number of new messages that have been received, the number of saved messages, the number of facsimile message, the number of data messages, and the number of voice messages. Other announcements, of course, could also be made at this time.

At step **258**, the owner then receives a recording of the owner's menu with the appropriate DTMF digit for each option. For instance, the DTMF digit "1" may be associated with playing a message, the DTMF digit "2" may be associated with an options menu, and the DTMF digit "*" may be associated with returning to a previous menu or terminating the call if no previous menu exists.

A DTMF digit is detected at step **260** and the appropriate action is taken based upon the digit received. Thus, if the digit is determined to be a "1" at step **264**, the owner can play a message at step **266**. At step **266**, the owner is preferably greeted with a menu giving the owner the options of playing or downloading new messages, saved messages, facsimile messages, data messages, or voice messages. As should be apparent to those skilled in the art, the owner may receive one or more menus at step **266** and the owner may enter one or more DTMF digits in order to play or download a particular message.

If, instead, the digit is determined to be a "2" at step **268**, then the owner receives an options menu at step **270**. With the options menu, the owner can enter or change certain parameters of the MSDS **10**. For instance, the owner can change his or her password, the owner can change the manner in which facsimile messages are displayed on the computer **32**, the owner can change the image file format from GIF to another format, the owner can select the file formats for the voice messages, as well as other options.

If the "*" DTMF digit is received, as determined at step **272**, then the owner is returned to a previous menu. The "*" digit is also used to terminate the call when the owner has returned to the initial menu. The "*" digit is therefore universally recognized by the MSDS **10** throughout the various menus as a command for returning to a previous menu.

If the owner enters a DTMF digit that is not being used by the MSDS **10**, the owner receives an indication at step **276** that the key is invalid and the owner is then again provided with the owner's menu at step **258**. When the owner does not enter a DTMF digit while the owner's menu is being played, as determined at step **260**, the menu will be replayed n times. Once the menu has been replayed n times, as determined at step **262**, then the call will be terminated at step **278**.

If the password is incorrect, as determined at step **254**, then the MSDS **10** checks whether the user has made more than "n" attempts at step **280**. If "n" attempts have not been made, then a password incorrect message will be displayed to the user at step **282** and the user will once again be prompted for the password at step **250**. When the user has made "n" attempts to enter the correct password, the MSDS **10** will play a failure message to the user at step **284** and then terminate the call at step **286**. The specific number "n" may be three so that the call is terminated after three failed attempts.

The owner's menu may be responsive to an additional number of DTMF digits and may be structured in other ways. For instance, separate DTMF digits may direct the owner to the respective types of messages, such as a facsimile message, data message, or voice message. Also, separate DTMF digits may direct the owner to a recording of new messages or to a recording of saved messages. Other variations will be apparent to those skilled in the art.

A more detailed diagram of the MSDS **10** is shown in FIG. **13**. As shown in the figure, a plurality of DID trunks **15** are received by an input/output device **17** and are then sent to a central processor **3**. The number of DID trunks **15** may be changed to any suitable number that would be necessary to accommodate the anticipated number of telephone calls that may be made to the MSDS **10**. The input/output device **17** routes a call on one of the DID trunks **15** to an open port of the central processor **3** and is preferably a DID Interface Box manufactured by Exacom.

The central processor **3** receives the calls on the DID trunks **15** and stores the messages in storage **11** in accordance with software **7**. Preferably, a separate directory in storage **11** is established for each user having an account on the MSDS **10** so that all of the messages for a single user will be stored in the same directory. It should be understood that the number of processors within the central processor **3** is dependent upon the number of DID trunks **15**. With a greater number of DID trunks **15** capable of handling a larger number of telephone calls, the central processor **3** may actually comprise a number of computers. The input/output device 17 would then function to route incoming calls to an available computer within the central processor 3.

A more detailed diagram of the central processor 3 is shown in FIG. 14. The central processor 3 comprises a telephone line interface 21 for each DID trunk 15. The telephone interface 21 provides the ringing sounds and other communication interfacing with the telephone lines. The signals from the telephone interface 21 are routed to a pulse/tone decoder 23 and to a digital signal processor (DSP) 25. The pulse/tone decoder 23 detects the address signal off of an incoming call and sends the address signal onto a bus 29 to a microprocessor 27. The DSP performs the necessary signal processing on the incoming calls and routes the processed signals to the microprocessor 27.

The microprocessor 27 will then read the address signal from the pulse/tone decoder 23 and store the message from the DSP 25 in an appropriate directory in storage 11. As discussed above, the central processor 3 may comprise a number of computers or, more precisely, a number of microprocessors 27 with each microprocessor 27 handling the calls from a certain number, such as four, DID trunks 15. The microprocessor 27 may comprise any suitable microprocessor, but is preferably at least a 486 PC.

In addition to handling incoming calls and storing the messages in storage 11, the central processor 3 also coordinates the interactive voice response system of the MSDS 10. The software 7 would incorporate the flowcharts of operations for receiving a message shown in FIG. 3, for detecting the type of message on an incoming call shown in FIG. 10, for receiving voice messages shown in FIG. 11, and for receiving an owner's call shown in FIG. 12. Based upon the above-referenced flowcharts and the respective descriptions, the production of the software 7 is within the capability of one of ordinary skill in the art and will not be described in any further detail.

The Internet Server 5 is connected to the central processor 3, such as through a local area network, and also has access to the storage 11. The Internet Server 5 performs a number of functions according to software 9. For instance, the Internet Server 5 retrieves the data files stored in storage 11 by the central computer 3 and converts the files into the appropriate HTML files. The converted HTML files are then stored in storage 11 and may be downloaded to the computer 32 through the Internet 30. The Internet Server 5 also handles the requests from the computer 32, which might require the retrieval of files from the storage 11 and possibly the generation of additional HTML files.

The software 9 for the Internet Server 5 would therefore incorporate the flowchart of operations for generating HTML files according to user preferences shown in FIG. 4, for generating requested information from a user shown in FIG. 5, for converting facsimile messages into HTML shown in FIG. 6, for converting voice messages into HTML shown in FIG. 8, and for converting data messages into HTML shown in FIG. 9. Based upon the above-referenced flowcharts and their respective descriptions, the production of the software 9 is within the capability of one of ordinary skill in the art and need not be described in any further detail.

Nonetheless, a more detailed block diagram of the Internet Server 5 is shown in FIG. 15. The Internet Server 5 runs on a suitable operating system (OS) 39, which is preferably Windows NT. The Internet Server 5 has a number of application programs 31, such as the ones depicted in the flowcharts discussed above, for communicating with the central processor 3 and for accessing data from storage 11 and also from memory 33.

The memory 33, inter alia, would contain the data indicating the preferences of each user. Thus, for example, when a facsimile message in the TIFF/F format is retrieved by the Internet Server 5, the Internet Server 5 would ascertain from the data in memory 33 the preferred option of displaying the facsimile message and would generate the appropriate HTML files.

All interfacing with the Internet 30 is handled by the HTTPD 37, which, in the preferred embodiment, is "Enterprise Server" from NetScape Communications Corp. Any requests from users, such as a request for a file, would be handled by the HTTPD 37, transferred through the CGI 35, and then received by the application programs 31. The application programs 31 would then take appropriate actions according to the request, such as transferring the requested file through the CGI 35 to the HTTPD 37 and then through the Internet 30 to the user's computer 32.

The Internet Server 5 may be connected to a paging system 13. Upon the arrival of a new message, in addition to sending an E-mail message to the user's mailbox, the Internet Server 13 may also activate the paging system 13 so that a pager 15 would be activated. In this manner, the user could receive almost instantaneous notification that a message has arrived.

The paging system 13 is preferably one that transmits alphanumeric characters so that a message may be relayed to the user's pager 15. The Internet Server 5 therefore comprises a signal processor 41 for generating signals recognized by the paging system 13 and a telephone interface 43. The signal processor 41 preferably receives information from the application programs 31 and generates a paging message in a paging file format, such as XIO/TAP. The telephone interface 43 would include a modem, an automatic dialer, and other suitable components for communicating with the paging system 13.

The information from the application programs 31 may simply notify the user of a message or may provide more detailed information. For instance, with a facsimile message, the information from the application programs 31 may comprise CSI information identifying the sender's telephone number. The user would therefore receive a message on the pager 15 informing the user that a facsimile message was received from a specified telephone number. The amount and type of information that may be sent to the user on the pager 15 may vary according to the capabilities of the paging system 13 and may provide a greater or lesser amount of information than the examples provided.

The Internet Server 5 is not limited to the structure shown in FIG. 15 but may comprise additional components. For instance, the HTTPD 37 would be linked to the Internet 30 through some type of interface, such as a modem or router. The Internet Server 5 may be connected to the Internet 30 through typical phone lines, ISDN lines, a T1 circuit, a T3 circuit, or in other ways with other technologies as will be apparent to those skilled in the art.

Furthermore, the Internet Server 5 need not be connected to the Internet 30 but may be connected to other types of networks. For instance, the Internet Server 5, or more generally the network Server 5, could be connected to a large private network, such as one established for a large corporation. The network Server 5 would operate in the same manner by converting messages into HTML files, receiving requests for information from users on the network, and by transmitting the information to the users.

Also, at least one interface circuit would be located between the Internet Server 5 and the central processor 3 in order to provide communication capabilities between the Internet Server **5** and the central processor **3**. This network interface may be provided within both the Internet Server **5** and the central processor **3** or within only one of the Internet Server **5** or central processor **3**.

Examples of the Internet Server **5** software layers are shown in FIGS. **16**(A) and **16**(B), with FIG. **16**(A) representing the Internet Server **5** in an asynchronous mode of communication and FIG. **16**(B) representing the Internet **5** in a synchronous mode of communication. As shown in the figures, the software **9** for the Internet Server **5** may additional comprise an Internet Deamon for running the HTTPD **37**. The software **9** for the Internet Server **5** would also include TCP/IP or other transport layers. Moreover, while the authentication is provided through the HTTPD **37**, the authentication of the user's password and ID may be supplemented or replaced with other ways of authentication.

The term synchronous has been used to refer to a mode of operation for the MSDS **10** in which the all possible HTML files for a message are generated at the time the message is received. The HTML files may be generated by the central processor **3** or by the application programs **31**. When a request for information is then later received by the HTTPD **37**, the information has already been generated and the HTTPD **37** only needs to retrieve the information from storage **11** and transmit the information to the user's computer **32**. With a synchronous mode of operation, the CGI **35** would be unnecessary.

The MSDS **10** preferably operates according to an asynchronous mode of operation. In an asynchronous mode of operation, information requested by the user may not be available and may have to be generated after the request. The asynchronous mode of operation is preferred since fewer files are generated, thereby reducing the required amount of storage **11**. Because the information requested by a user may not be available, some anchors cannot specify the filename, such as "2.html," but will instead contain a command for the file. For instance, an anchor may be defined as<AHREF="/faxweb/users/2496801/ viewpage.cgi?FAX_NUM=1&PAGE=1&VIEW_MODE=FULL"> for causing the CGI **35** to run a viewpage program so that page 1 of facsimile message 1 will be displayed in a full size image. The CGI **35** will generate the requested information when the information has not been generated, otherwise the CGI **35** will retrieve the information and relay the information to the HTTPD **37** for transmission to the user.

With the invention, the MSDS **10** can reliably receive voice, facsimile, and data messages for a plurality of users and can receive more than one message for a user at a single time. The messages are stored by the MSDS **10** and can be retrieved at the user's convenience at any time by connecting to the Internet **30**. The Internet World Wide Web **30** is a constantly expanding network that permits the user to retrieve the messages at virtually any location in the world. Since the user only needs to incur a local charge for connecting to the Internet **30**, the user can retrieve or review messages at a relatively low cost.

Even for the user's at the office or at home, the MSDS **10** provides a great number of benefits. The user would not need a facsimile machine, voice mail system, or a machine dedicated for receiving data messages. The user also need not worry about losing part of the message or violating the confidential nature of the messages. The user, of course, can still have a facsimile machine and dedicated computer for data messages. The MSDS **10**, however, will permit the user to use the telephone company's "call forwarding" feature so that messages may be transferred to the MSDS **10** at the user's convenience, such as when the user is away from the office.

The software **7** and software **9** are not limited to the exact forms of the flowcharts shown but may be varied to suit the particular hardware embodied by the invention. The software may comprise additional processes not shown or may combine one or more of the processes shown into a single process. Further, the software **7** and **9** may be executed by a single computer, such as a Silicon Graphics Workstation, or may be executed by a larger number of computers.

The facsimile messages preferably undergo signal processing so that the images of the facsimile messages are converted from a two tone black or white image into an image with a varying gray scale. As is known in the art, a gray scale image of a facsimile message provides a better image than simply a black or white image of the message. The signal processing may comprise any suitable standard contrast curve method of processing, such as anti-aliasing or a smoothing filter. The signal processing may occur concurrently with the conversion from TIFF/F to GIF and is preferably performed for both full and reduced size images of the facsimile messages.

Furthermore, the user may be provided with a greater or fewer number of options in displaying or retrieving messages. The options are not limited to the exact forms provided but may permit the user to review or retrieve the messages in other formats. The options may also permit a user to join two or messages into a single message, to delete portions of a message, or to otherwise the contents of the messages. Also, the various menus provided to the user over the telephone may have a greater number of options and the MSDS **10** may accept responses that involve more than just a single DTMF digit.

The specific DTMF digits disclosed in the various menus are only examples and, as will be apparent to those skilled in the art, other digits may be used in their place. For instance, a "9" may be used in the place of a "*" in order to exit the menu or to return to a previous menu. Also, the DTMF digits may be changed in accordance with the user's personal convention. If the user had a previous voice mail system, the user could customize the commands to correspond with the commands used in the previous system in order to provide a smooth transition to the MSDS **10**.

The MSDS **10** may restrict a user to only certain types of messages. For instance, a user may want the MSDS **10** to store only facsimile messages in order to reduce costs of using the MSDS **10**. In such a situation, the MSDS **10** would perform an additional step of checking that the type of message received for a user is a type of message that the MSDS **10** is authorized to receive on the user's behalf. When the message is an unauthorized type of message, the MSDS **10** may ignore the message entirely or the MSDS **10** may inform the user that someone attempted to send a message to the MSDS **10**.

Moreover, the MSDS **10** has been described as having the central processor **3** for handling incoming calls and the Internet Server **10** for interfacing with the Internet **30**. The invention may be practiced in various ways other than with two separate processors. For instance, the central processor **3** and the Internet Server **5** may comprise a single computer or workstation for handling the incoming calls and for interfacing with the Internet **30**. The MSDS **10** may convert the messages into HTML files prior to storing the messages. Also, the central processor **3** may communicate with the paging system **13** instead of the Internet Server **5**.

Additionally, as discussed above, the central processor **3** may comprise a number of microprocessors **27** for handling a large number of DID trunks.

The invention has been described as converting the messages into HTML and transmitting the HTML files over the Internet **30** to the computer **32**. The HTML format, however, is only the currently preferred format for exchanging information on the Internet **30** and is actually only one type of a Standard Generalized Mark-Up Language. The invention is therefore not limited to the HTML format but may be practiced with any type of mixed media page layout language that can be used to exchange information on the Internet **30**.

SGML is not limited to any specific standard but encompasses numerous dialects and variations in languages. One example of an SGML dialect is virtual reality mark-up language (VRML) which is used to deliver three dimensional images through the Internet. As another example, the computer **32** for accessing the MSDS **10** through the Internet **30** may comprise a handheld device. A handheld device is generally characterized by a small display size, limited input capabilities, limited bandwidth, and limited resources, such as limited amount of memory, processing power, or permanent storage. In view of these limited capabilities, a handheld device markup language (HDML) has been proposed to provide easy access to the Internet **30** for handheld devices. The SGML information transmitted by the MSDS **10** to the computer **32** may therefore comprise HDML information suitable for a handheld device or may comprise VRML.

As another example, Extensible Mark-Up Language (XML) is an abbreviated version of SGML, which makes it easier to define document types and makes it easier for programmers to write programs to handle them. XML omits some more complex and some less-used parts of the standard SGML in return for the benefits of being easier to write applications for, easier to understand, and more suited to delivery and inter-operability over the Web. Because XML is nonetheless a dialect of SGML, the MSDS **10** therefore encompasses the translation of facsimile, voice, and data messages into XML, including all of its dialects and variations, and the delivery of these messages to computers **32** through the Internet **30**.

As a further example, the MSDS **10** encompasses the use of "dynamic HTML." "Dynamic HTML" is a term that has been used to describe the combination of HTML, style sheets, and scripts that allows documents to be animated. The Document Object Model (DOM) is a platform-neutral and language neutral interface allowing dynamic access and updating of content, structure, and style of documents. The MSDS **10** may therefore include the use of the DOM and dynamic HTML to deliver dynamic content to the computer **32** through the Internet **30**.

The MSDS **10** is also not limited to any particular version or standard of HTTP and thus not to any particular hypertext transfer protocol deamon **37**. In general, HTTP is a data access protocol run over TCP and is the basis of the World Wide Web. HTTP began as a generic request-response protocol, designed to accommodate a variety of applications ranging from document exchange and management to searching and forms processing. Through the development of HTTP, the request for extensions and new features to HTTP has exploded; such extensions range from caching, distributed authoring and content negotiation to various remote procedure call mechanisms. By not having a modularized architecture, the price of new features has been an overly complex and incomprehensible protocol. For instance, a Protocol Extension Protocol (PEP) is an extension mechanism for HTTP designed to address the tension between private agreement and public specification and to accommodate extension of HTTP clients and servers by software components. Multiplexing Protocol (MUX) is another extension that introduces asynchronous messaging support at a layer below HTTP. As a result of these drawbacks of HTTP, a new version of HTTP, namely HTTP-NG, has been proposed and its purpose is to provide a new architecture for the HTTP protocol based on a simple, extensible distributed object-oriented model. HTTP-NG, for instance, provides support for commercial transactions including enhanced security and support for on-line payments. Another version of HTTP, namely S-HTTP, provides secure messaging. The MSDS **10** and the HTTPD **37** may incorporate these versions or other versions of HTTP.

In addition to different versions of HTTP, the HTTPD **37** of the MSDS **10** may operate with other implementations of HTTP. For instance, the W3C's has an implementation of HTTP called "Jigsaw." Jigsaw is an HTTP server entirely written in Java and provides benefits in terms of portability, extensibility, and efficiency. The MSDS **10** may employ Jigsaw or other implementations of HTTP.

With regard to the transmission of messages to the user's computer **32**, the MSDS **10** permits the user to sample the voice message or to preview the facsimile message without requiring the MSDS **10** to transmit the entire message to the computer **32**. This sampling ability is a significant benefit since the transmission of the entire message would frequently tie up the computer **32** for a rather long period of time. Thus, with the preview or sample feature, the user can determine whether the user needs the message transmitted to the computer **32**.

If the user does decide that the entire message needs to be transmitted, as stated above, the user's computer **32** might be receiving the message for a relatively long period of time. After the entire message has been received, the user then has the options of viewing, listening, retrieving, or saving the message. As an alternative, the user's computer may instead indicate the contents of the message to the user as the message is being received.

For instance, with a voice message, the user's computer **32** could send the message to an audio speaker as the message is being received. In this manner, the message would be played in real time and the user would not need to wait until the entire message is received before listening to the message. In order to play the messages in real time, the messages are preferably in the RealAudio (RA) format, which the user can select as the preferred file format for voice messages.

In operation, the MSDS **10** would transmit an HTML file containing an RA file. If the user selects the RA file with the browser on the computer **32**, the browser will activate a program for use with RA files. The operations and functioning of this program will be apparent to those skilled in the art and will be available as a separate software package or will be incorporated within a browser program. The RA program will request the RA data file containing the message from the MSDS **10** and, as the RA file is being received at the computer **32**, this program will play the message in real time.

The MSDS **10** and the user's computer **32** could also be arranged so that each page or even line of a facsimile message could be displayed as the computer **32** receives the facsimile message. Further, although the transmission of a data message is relatively fast in comparison to a voice or facsimile message, the computer **32** could also be programmed to permit access to the data message as the message is being received.

The invention has been described as storing and transmitting voice messages. It should be understood that the voice message would probably be the most often type of audio message stored at the MSDS **10**. The invention, however, may be used with any type of audio message and is in no way limited to just voice messages.

According to another aspect of the invention, the MSDS **10** may be used as a file repository serving as an archive for a particular user or group of users. As described above, the MSDS **10** may maintain a list of all messages for a particular user which is displayed to the user when the user access his or her mailbox. The MSDS **10** may store all messages, whether they are voice, facsimile, or data, for a user in the database indefinitely. The MSDS **10** may therefore be relied upon by a user to establish the authenticity of a message and the existence or absence of a particular message. Through the MSDS **10**, a user can therefore maintain an accurate record of all received email messages, facsimile messages, and data transfers.

In addition to serving as a file depository, the MSDS **10** may also function as a document management tool. As described above with reference to FIG. **2**, when the MSDS **10** receives a message, the MSDS **10** updates a database with information on the message. This information includes the type of message, whether it is a facsimile message, voice message, or data message, the time and date at which the message was received, the size of the file, such as in bytes, the telephone number of the caller leaving the message, as well as other information, such as the number of pages of a facsimile message. Because the telephone number called is unique for each user, the information also includes the intended recipient of the message.

An example of a data entry **300** in storage **11** for a message is shown in FIG. **17**. The data entry **300** represents the entry for just a single message with each message having a separate data entry **300**. Preferably, the data entries **300** are stored in a relational database and may be searched through a structured query language (SQL).

As shown in FIG. **17**, the data field **300** for a message may comprise numerous data fields for describing the message. One of these data fields may comprise a field **301** for indicating the name of the person receiving the message. As will be appreciated by those skilled in the art, the person may be identified in numerous ways, such as by a portion of the person's name or by a unique number. Another field **302** in the data entry **300** indicates the type of the document, such as whether the document is a facsimile message, voice message, or data transfer, and fields **303** and **304** respectively indicate the date and time that the message was received by the MSDS **10**. The telephone number of the caller is indicated in field **305** while the size of the message, which may be measured in bytes, is indicated in field **306** and the number of pages of the message is indicated in field **307**. A document number for uniquely identifying the message is indicated in field **308**. As discussed above, the files or messages received for a particular user may be numbered sequentially in the order that they are received by the MSDS **10**. The files and messages, however, may be numbered or identified in other ways, such as by a combination of numbers with an identifier for the date when the message was received. Also, the documents number or identifier may be unique for each file or message directed to a user or, alternatively, may be unique for each file or message directed to a plurality of users, which is advantageous when the MSDS **10** tracks documents for an entire company or other group of users.

In addition to fields **301** to **308**, the data entry **300** for a message or file may have other fields **309** for describing or documenting the message or file. The other fields **309**, for instance, may be used to identify the type of storage that a message should receive. The messages or files may have different lengths of time that the message is stored before being automatically deleted. The type of storage, such as whether the full text of the message is stored, may also be indicated by field **309**. Another example of a trait that may be contained within the other field **309** is security. At times, a user may desire and may be granted access to another person's mailbox, such when the MSDS **10** tracks documents for an entire company. By designating a message or file as secure in field **309**, a user may restrict or deny access to that message or file by other users. The other fields **309** may also be used by a user to customize the MSDS **10** according to his or her own desires. For instance, if the user is a company, the company may want to classify messages according to the division at which the message is directed, such as one code for marketing, one for sales, one for engineering, and one for legal.

As another example of a use of one of the other fields **309**, a user can input notes in the other field **309**. When a user initially receives a data entry **300**, the entry **300**, for instance, may include data in all fields **301** to **308** except field **309**, which has been left blank. The user can then input his or her notes in the other field. An initial data entry **300** may include the field **305** for the caller's telephone number which contains the digits for the calling number. The user, however, may not readily recognize the caller from just reading the telephone number listed in field **305**. To more clearly indicate the caller, the user may input notes in field **309** to identify the caller's name. Alternatively, the notes in field **309** may reflect part or all of the contents of the message. The user may receive a large document or message and may input a brief description of the document or message in the field **309**. As another example, the recipient of the message may read the message or document and discover that the caller is requesting some service or goods from the recipient, such as a request for certain documents or delivery of a certain quantity of goods. The recipient may read the document or message and place somes notes in the field **309** to indicate the type of follow-up service or action that needs to be taken. An assistant to the recipient can then view the notes in field **309** and take appropriate steps to ensure that the requested service or goods are delivered. If the data entry is security protected, one of the other fields **309**, as discussed above, may grant the assistant limited access to just the field **309** or may grant more expansive access whereby the assistant can view fields **301** to **309** as well as the actual document or message. The fields **309** may serve various other purposes, as will be apparent to those skilled in the art.

FIG. **18** illustrates a process **320** for using the MSDS **10** for document management purposes. With reference to FIG. **18**, a user sends a search request to the MSDS **10** for a particular document or set of documents at step **321**. The user may issue this request with the computer **32** by clicking on a link, such as a link to "Search Documents," which may be presented to the user by the MSDS **10** after the user has been granted accesses to his or her mailbox at step **72** shown in FIG. **3**. The MSDS **10** may present the user with the option to search the document archives at other times, such as when the user first attempts to access the mailbox at step

62, or when the URL received by the HTTPD 37 from computer 32 points toward the document archives.

In response to this request, the HTTPD 37 sends the user a search query form at step 322 to allow the user to define a desired search. An example of a search query form is shown in FIG. 19. The search query form may include an entry for each of the data fields 301 to 309 in the data entry 300. For instance, the user may input one or more names for a recipient and have the MSDS 10 search for all messages or files directed to just those recipients. The user may also indicate the type of document, such as whether it is a facsimile, voice message or data file. The search query form also has entries for the date or time, which preferably accept ranges of times and dates, and an entry for the telephone number of the caller to the MSDS 10. The search query form may also include an entry for the size of the file or for the number of pages, which is relevant if the message is a facsimile message. The search query form may also include an entry for the document number, which may accept a range of document numbers, and also an entry for another field.

At step 323, the user enters the search parameters in the search query form with computer 32 and returns the information to the MSDS 10 through the Internet 30. The user may define the search about any one data field or may define the search about a combination of two or more data fields. For instance, as reflected in the completed search query form shown in FIG. 20, a user may define a search by designating the document type as a facsimile and the calling number as (404) 249-6801. Once the user has finished defining the search, the user then selects the "SEARCH" link shown at the bottom of the screen whereby the user's computer 32 would send the completed search query form through the Internet 30 to the HTTPD 37 of the MSDS 10.

At step 324, the HTTPD 37 receives the completed search query form and, through CGI 35, invokes one or more of the application programs 31 for performing the desired search for any files or messages falling within the parameters of the search. The results of the search are passed from the application programs 31 through the CGI 35 to the HTTPD 37 and, at step 325, are returned to the user through the Internet 37. Preferably, the MSDS 10 returns the search results in the form of a listing of all files or messages contained within the search parameters, although the MSDS 10 may return the results in other ways.

An example of the search results of the query shown in FIG. 20 is shown in FIG. 21. As discussed above, the parameters of the search were all facsimile messages from telephone number (404) 249-6081. With reference to FIG. 21, this query resulted in three messages being discovered. The first document has a document number 11 and is described as being a facsimile from the designated telephone number to Jane Doe on May 31, 1995, and consists of three pages. This first-listed document is an example of the facsimile shown in FIG. 7. The other two documents respectively correspond to document numbers 243 and 1,002 and are also from the designated telephone number.

At step 326, the user selects the desired file or message from the listing of messages and files. For instance, by clicking on the first listed document, namely document number 11, the computer 32 sends a request to the MSDS 10 for a viewing of that document and, in response, the MSDS 10 provides a viewing of the document according to the user defined preferences. As described above, the user may receive a reduced size image of the first page, a full size image of the first page, reduced size images of all pages, or full size images of all pages of the facsimile message. Thus, if the user selected the fourth display option as the user defined preference, the MSDS 10 would return an image of the first page of the facsimile, such as the one depicted in FIG. 7.

At step 326, the user may also have the MSDS 10 save the search results. For instance, as shown in FIG. 21, the user may input the name of "CHARLES R. BOBO FACSIMILES" as the name for the search. By clicking on the "SAVE SEARCH AS" link, the name of the search is provided from the computer 32 to the MSDS 10. At the MSDS 10, the HTTPD 37 transfers the information from the computer 32 to the CGI 35 and the CGI 35 invokes an application program 31 to store the results of the search in storage 11 under the designated name. The invoked application program 31 preferably does not store the contents of all messages but rather stores a listing of the search results in the storage 11.

The results of a search may be stored in storage 11 as either a closed search or an open search. If the MSDS 10 saves the results of a search as an open search, then the files or messages in that named search may be updated with recent files or messages falling within the particular search parameters for the search. On the other hand, a closed search is one in which the files or messages in the named search are limited to those existing at the time of the search. For example, if the MSDS 10 saved the search results shown in FIG. 21 as a closed search, then any retrieval of the "CHARLES R. BOBO FACSIMILES" would result in only the three listed documents. If, on the other hand, the search named as the "CHARLES R. BOBO FACSIMILES" was saved by the MSDS 10 as an open search, then the MSDS 10 would reactivate the search query shown in FIG. 20 in response to a request by the computer 32 for that search in order to obtain all facsimile messages from that particular telephone number, including those received after the initial saving of the search results.

With reference to FIG. 19, rather than defining a new search, the user may click on the "STORED SEARCHES" link in order to receive the results of a previously performed search. For example, by clicking on this link, the MSDS 10 may return a listing of searches stored for that particular user, such as the searches shown in FIG. 22. As shown in this figure, the "CHARLES R. BOBO FACSIMILES" is included within the list of stored searches. If the user then selected the "CHARLES R. BOBO FACSIMILES" search, the user may then be presented with the listing of facsimiles shown in FIG. 21, possibly including recent additions to the search group.

With reference to FIG. 19, the MSDS 10 may also provide a user with a link to "RECENT FILES" at step 322. By selecting this link, the MSDS 10 may return a listing of all facsimile, voice, and data messages received with a particular period of time, such as the last month. By placing the "RECENT FILES" link on the search query form rather than in the listing of "STORED SEARCHES," the user can quickly turn to the most recent files and messages. The search query form may contain other such easy-access links, such as a link to the last search performed by the MSDS 10 on behalf of the user.

The messages or files received by the MSDS 10 need not arrive from a third party. In other words, the MSDS 10 may be used as a file repository or as a file manager for documents generated by the user itself. The user may call the designated telephone number for receiving messages and transmit voice messages, data messages, or facsimile messages and have the MSDS 10 document the receipt and content of these messages. A user may easily use a facsimile machine as a scanner for entering documents into the storage **11** of the MSDS **10**.

The MSDS **10** may have applications in addition to those discussed-above with regard to serving as a message deliverer, file repository, and file manager. For instance, the MSDS **10** may perform some additional processing on the incoming calls prior to forwarding them to the user. For voice messages, this processing may involve transcribing the message and then returning the transcribed messages to the user. The MSDS **10** may therefore be viewed as offering secretarial assistance which may be invaluable to small companies or individuals who cannot afford a secretary or even to larger businesses who may need some over-flow assistance. The transcription may be provided by individuals located in any part of the world or may be performed automatically by a speech-to-text recognition software, such as VoiceType from IBM.

Another type of processing that the MSDS **10** may provide is translation services. The incoming call, whether it is a voice, facsimile, or data message, can be converted into SGML and then forwarded first to a translator. Given the reach of the Internet, the translator may be located virtually anywhere in the world and can return the translated document via the Internet to the MSDS **10**. The MSDS **10** can notify the user that the translation has been completed through email, voice mail, pager, facsimile, or in other ways. The user would then connect to the Internet and retrieve the translated document. The translation services of the MSDS **10** may also provide transcription of the message, such as with speech-to-text recognition software.

The foregoing description of the preferred embodiments of the invention have been presented only for the purposes of illustration and description. It is not intended to be exhaustive or to limit the invention to the precise form disclosed. Many modifications and variations are possible in light of the above teaching.

The embodiments were chosen and described in order to explain the principles of the invention and their practical application so as to enable others skilled in the art to utilize the invention and various embodiments and with various modifications as are suited to the particular use contemplated. It is intended that the scope of the invention only be limited by the claims appended hereto.

I claim:

1. A network message storage and delivery system, comprising:

means for receiving an incoming call and for detecting an address signal associated with said incoming call, said address signal associated with a user of said message storage and delivery system;

means for receiving a message accompanied with said address signal, said message being in a first file format;

means for converting said message from said first file format to a second file format;

means for storing said message in said second file format in a storage area;

means for receiving a request from said user for said message and for retrieving said message from said storage area; and

means for transmitting a least a portion of said message in said second file format to said user over a transmission link;

wherein said portion of said message is transmitted to said user over the network, said second file format is a mixed media page layout language and comprises a standard generalized mark-up language.

2. A network message storage and delivery system, comprising:

means for receiving an incoming call and for detecting an address signal associated with said incoming call, said address signal associated with a user of said message storage and delivery system;

means for receiving a message accompanied with said address signal, said message being in a first file format;

means for converting said message from said first file format to a second file format;

means for storing said message in said second file format in a storage area;

means for receiving a request from said user for said message and for retrieving said message from said storage area; and

means for transmitting a least a portion of said message in said second file format to said user over a transmission link;

wherein said portion of said message is transmitted to said user over the network, said second file format is a mixed media page layout language, and said network comprises the Internet.

3. A network message storage and delivery system, comprising:

a central processor for receiving an incoming call, for detecting an address signal on said incoming call, for detecting a message on said incoming call, and for placing said message in a storage area, said address signal being associated with a user of said network message storage and delivery system;

a network server for receiving said message from said storage area, for converting said message into a mixed media page layout language, and for placing said message in said storage area;

wherein when said network server receives a request from said user over said network, said network server transmits at least a portion of said message over said network to said user over a transmission link and wherein said network comprises the Internet and said network server comprises an Internet server.

4. A method of storing and delivering a message for a user, comprising the steps of:

receiving an incoming call and detecting an address signal associated with said incoming call, said address signal associated with a user;

receiving a message associated with said address signal, said message being in a first file format;

converting said message from said first file format to a second file format;

storing said message in said second file format in a storage area;

receiving a request from said user for said message and retrieving said message from said storage area; and

transmitting at least a portion of said message in said second file format to said user over a transmission link;

wherein said step of transmitting occurs over a network, said step of converting said message converts said message into a mixed media page layout language, and said step of transmitting occurs over the Internet.

* * * * *

US006658463B1

(12) **United States Patent**
Dillon et al.

(10) **Patent No.: US 6,658,463 B1**
(45) **Date of Patent: Dec. 2, 2003**

(54) **SATELLITE MULTICAST PERFORMANCE ENHANCING MULTICAST HTTP PROXY SYSTEM AND METHOD**

(75) Inventors: **Douglas M. Dillon**, Gaithersburg, MD (US); **T. Paul Gaske**, Rockville, MD (US)

(73) Assignee: **Hughes Electronics Corporation**, El Segundo, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/498,936**

(22) Filed: **Feb. 4, 2000**

Related U.S. Application Data

(60) Provisional application No. 60/138,496, filed on Jun. 10, 1999.

(51) **Int. Cl.**^{7} **G06F 15/16**

(52) **U.S. Cl.** **709/219**; 709/217; 709/203; 709/228

(58) **Field of Search** 709/217, 219, 709/202, 203, 204; 707/1, 8, 100, 200, 201

(56) **References Cited**

U.S. PATENT DOCUMENTS

6,055,364 A	*	4/2000	Speakman et al.	709/229
6,085,193 A	*	7/2000	Malkin et al.	709/200
6,128,655 A	*	10/2000	Fields et al.	709/219
6,463,447 B2	*	10/2002	Marks et al.	709/228

OTHER PUBLICATIONS

"Transparent Power–on Control of Token Ring and Token Bus File Server Computers", IBM Technical Disclosure Bulletin, Mar. 1995, vol. 38, Issue 3, pp. 55–56.*

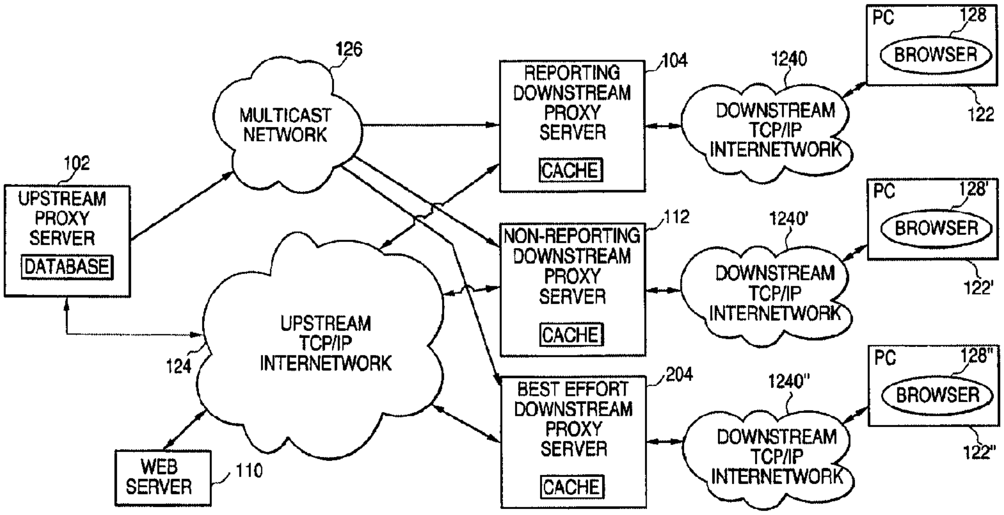
* cited by examiner

Primary Examiner—David Wiley
Assistant Examiner—Joseph E. Avellino
(74) *Attorney, Agent, or Firm*—John T. Whelan; Michael Sales

(57) **ABSTRACT**

A communication system including an upstream proxy server and two reporting downstream proxy servers, where the upstream proxy server is capable of multicasting a uniform resource locator (URL) to the reporting downstream proxy servers, the reporting downstream proxy servers interact with the upstream proxy server to resolve cache misses and the upstream proxy servers returns a resolution to a cache miss via multicast. A downstream proxy server which filters multicast transmissions of URLs and stores a subset of the URLs for subsequent transmission, where relative popularity is used to determine whether to store a multicast URL. An upstream proxy server capable of multicasting URLs to reporting downstream proxy servers, where the upstream proxy server interacts with the two reporting downstream proxy servers to resolve cache misses and the upstream proxy server returns a resolution to the cache misses via multicast. A proxy server protocol which includes a transaction request further including a request header, request content, and a request extension that supports multicast hit reporting and a transaction response further including a response header, response content, and a response extension which supports multicast cache pre-loading. A transaction response header which includes a popularity field indicating the popularity of a global name with respect to other global names and an expiration field indicating an expiration of the global name.

20 Claims, 12 Drawing Sheets


126
MULTICAST NETWORK
102
UPSTREAM PROXY SERVER
DATABASE
UPSTREAM TCP/IP INTERNETWORK
124
WEB SERVER
110
REPORTING DOWNSTREAM PROXY SERVER
CACHE
104
NON-REPORTING DOWNSTREAM PROXY SERVER
CACHE
112
BEST EFFORT DOWNSTREAM PROXY SERVER
CACHE
204
1240
DOWNSTREAM TCP/IP INTERNETWORK
1240'
DOWNSTREAM TCP/IP INTERNETWORK
1240"
DOWNSTREAM TCP/IP INTERNETWORK
PC
128
BROWSER
122
PC
128'
BROWSER
122'
PC
128"
BROWSER
122"

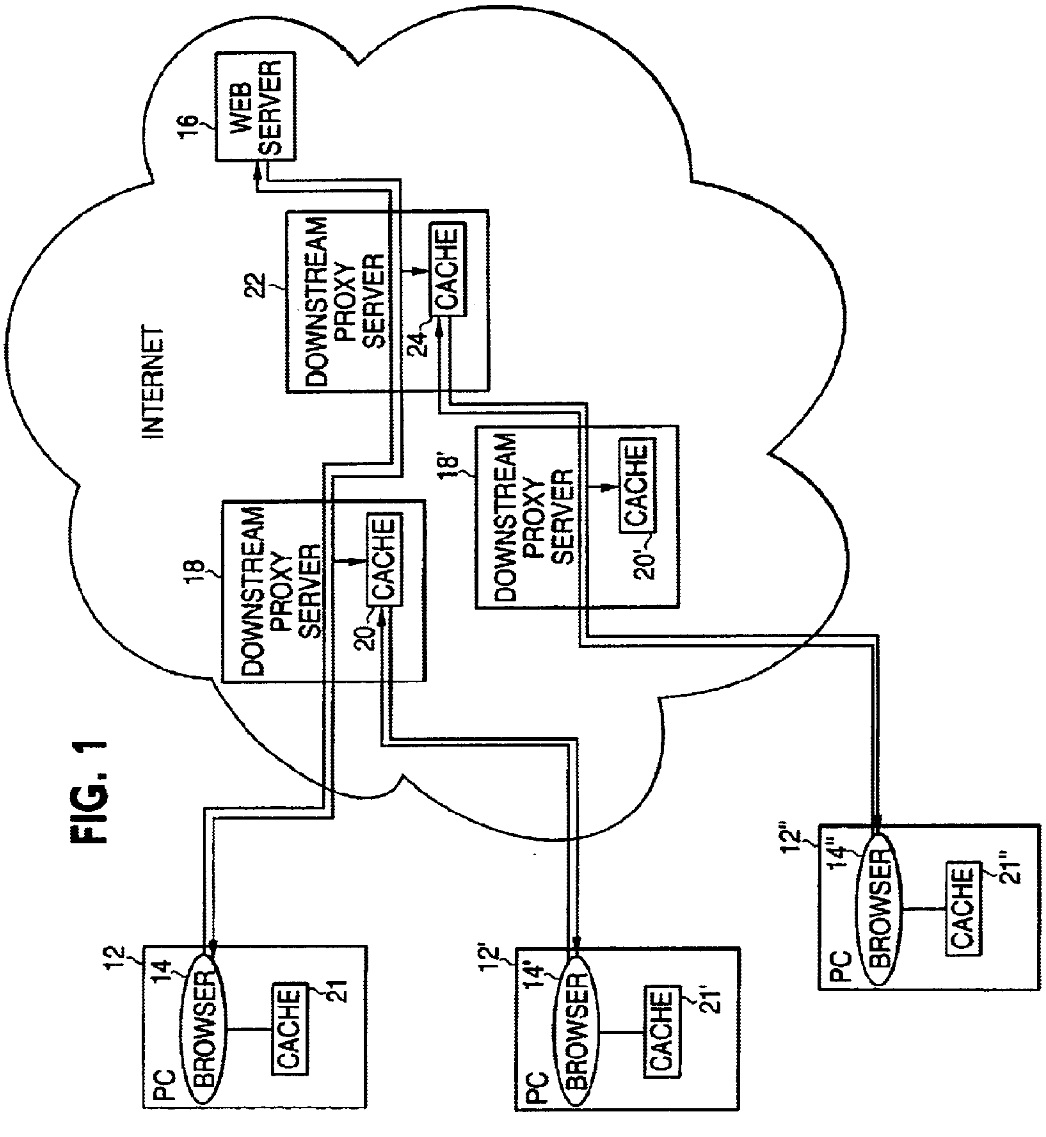

INTERNET
16
WEB SERVER
22
DOWNSTREAM PROXY SERVER
CACHE
24
18
DOWNSTREAM PROXY SERVER
CACHE
20
18'
DOWNSTREAM PROXY SERVER
CACHE
20'
12
PC
14
BROWSER
CACHE
21
12'
PC
14'
BROWSER
CACHE
21'
12"
PC
14"
BROWSER
CACHE
21"

FIG. 1

FIG. 2

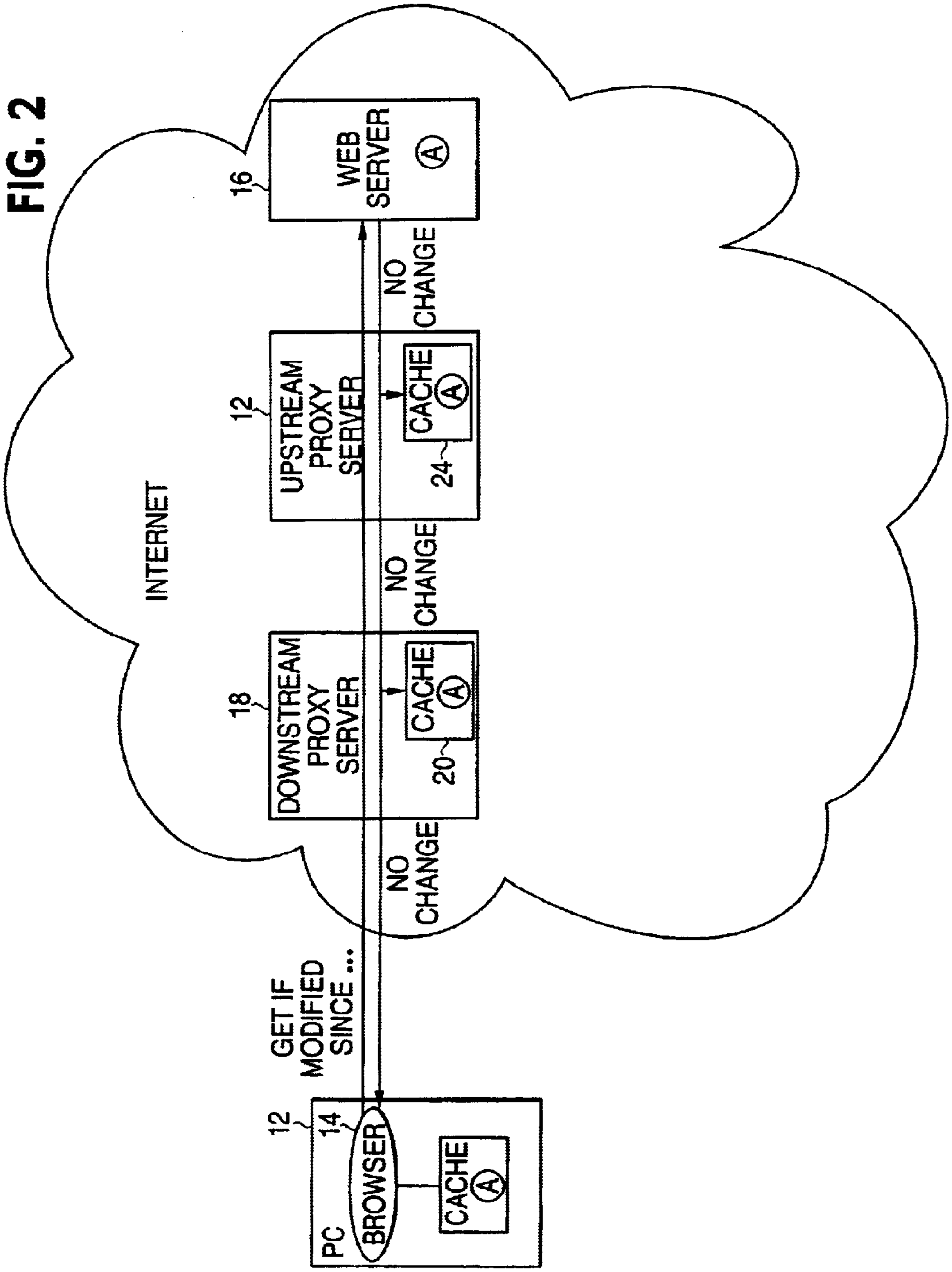

PC
BROWSER
CACHE
Ⓐ
12
14
GET IF MODIFIED SINCE ...
NO CHANGE
DOWNSTREAM PROXY SERVER
18
CACHE
Ⓐ
20
NO CHANGE
INTERNET
UPSTREAM PROXY SERVER
12
CACHE
Ⓐ
24
NO CHANGE
WEB SERVER
Ⓐ
16

FIG. 3

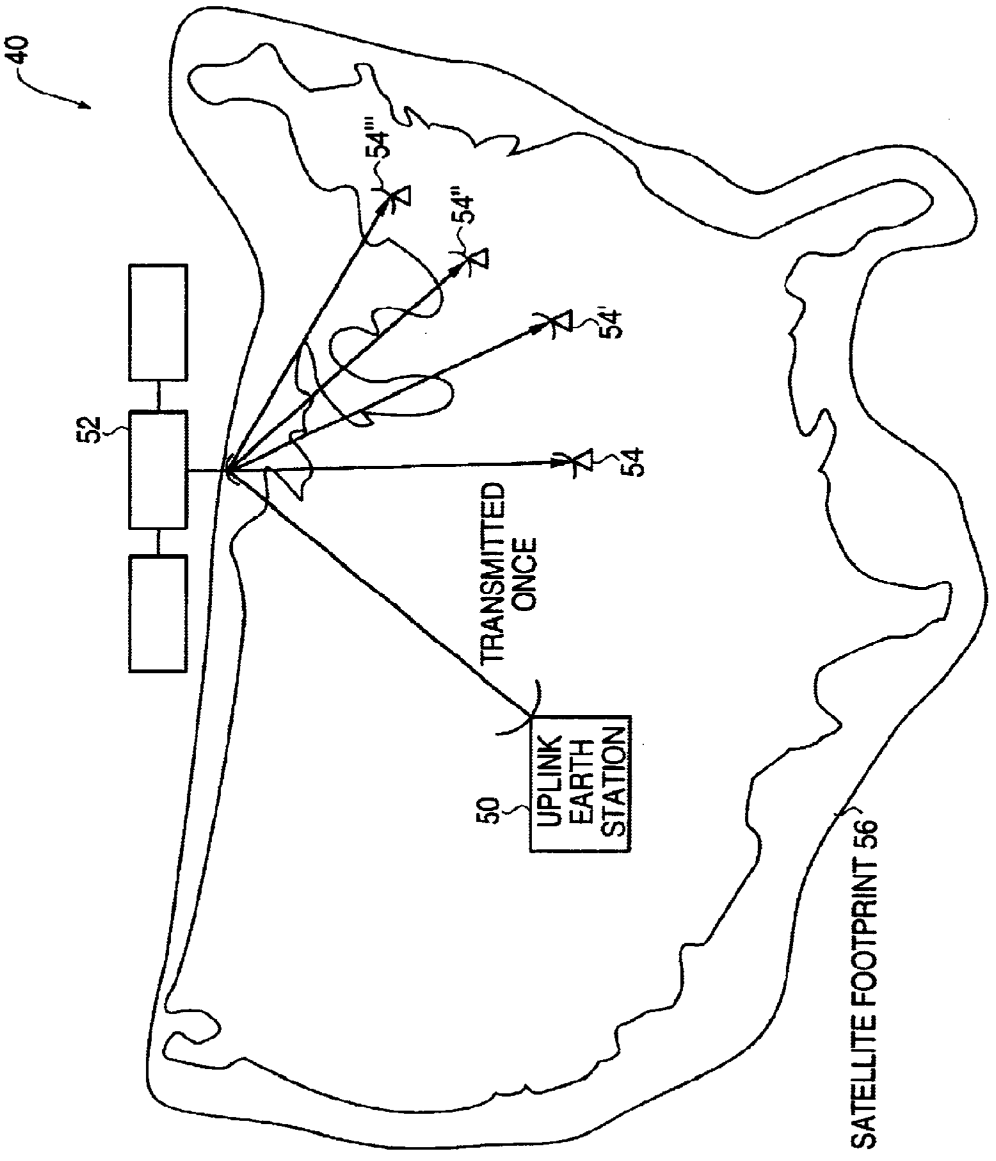
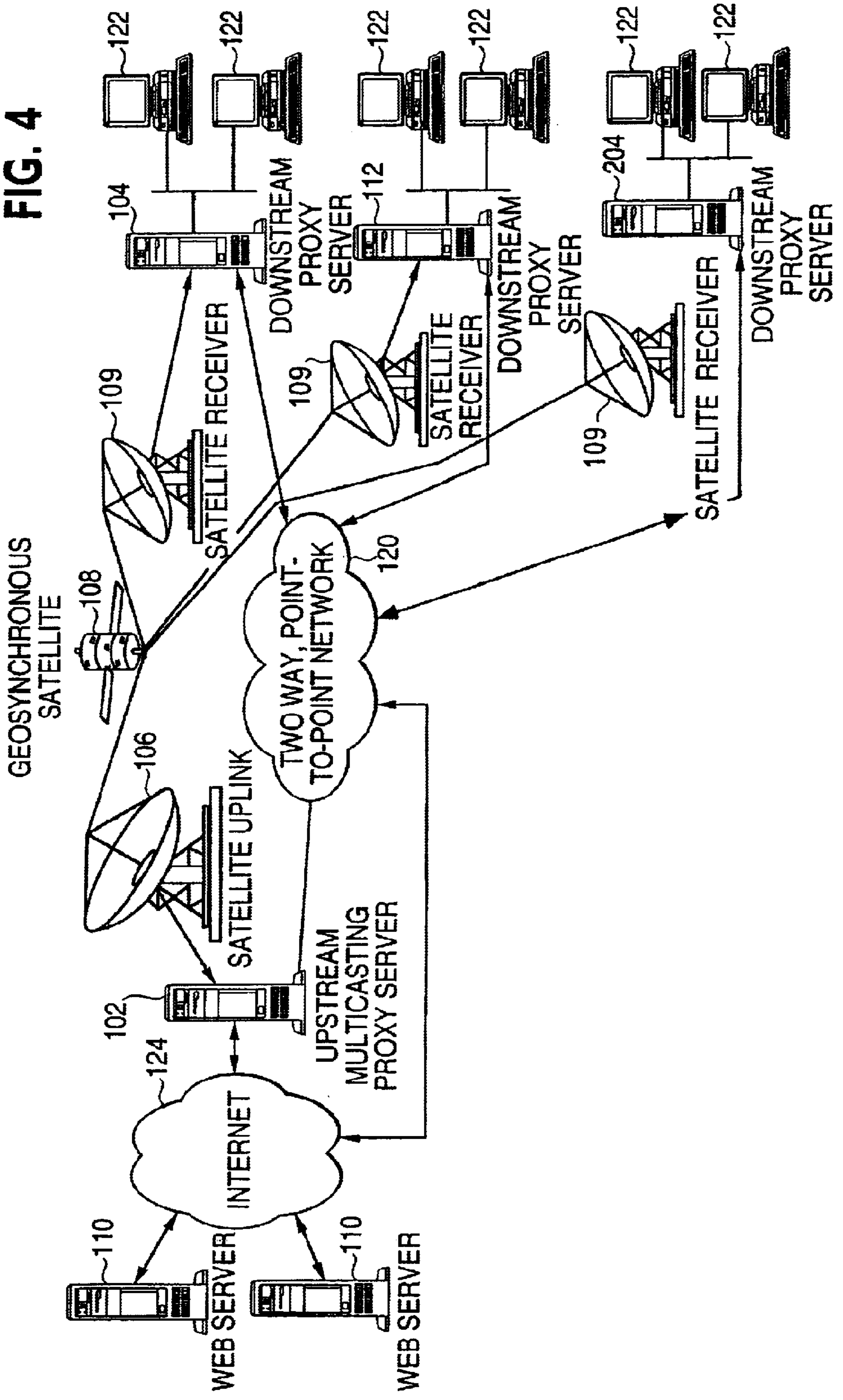

40
52
54'''
54''
54'
54
TRANSMITTED ONCE
50
UPLINK EARTH STATION
SATELLITE FOOTPRINT 56

FIG. 4


GEOSYNCHRONOUS SATELLITE
108
106
SATELLITE UPLINK
102
UPSTREAM MULTICASTING PROXY SERVER
124
INTERNET
110
WEB SERVER
110
WEB SERVER
TWO WAY, POINT-TO-POINT NETWORK
120
109
SATELLITE RECEIVER
104
DOWNSTREAM PROXY SERVER
122
122
109
SATELLITE RECEIVER
112
DOWNSTREAM PROXY SERVER
122
122
109
SATELLITE RECEIVER
204
DOWNSTREAM PROXY SERVER
122
122

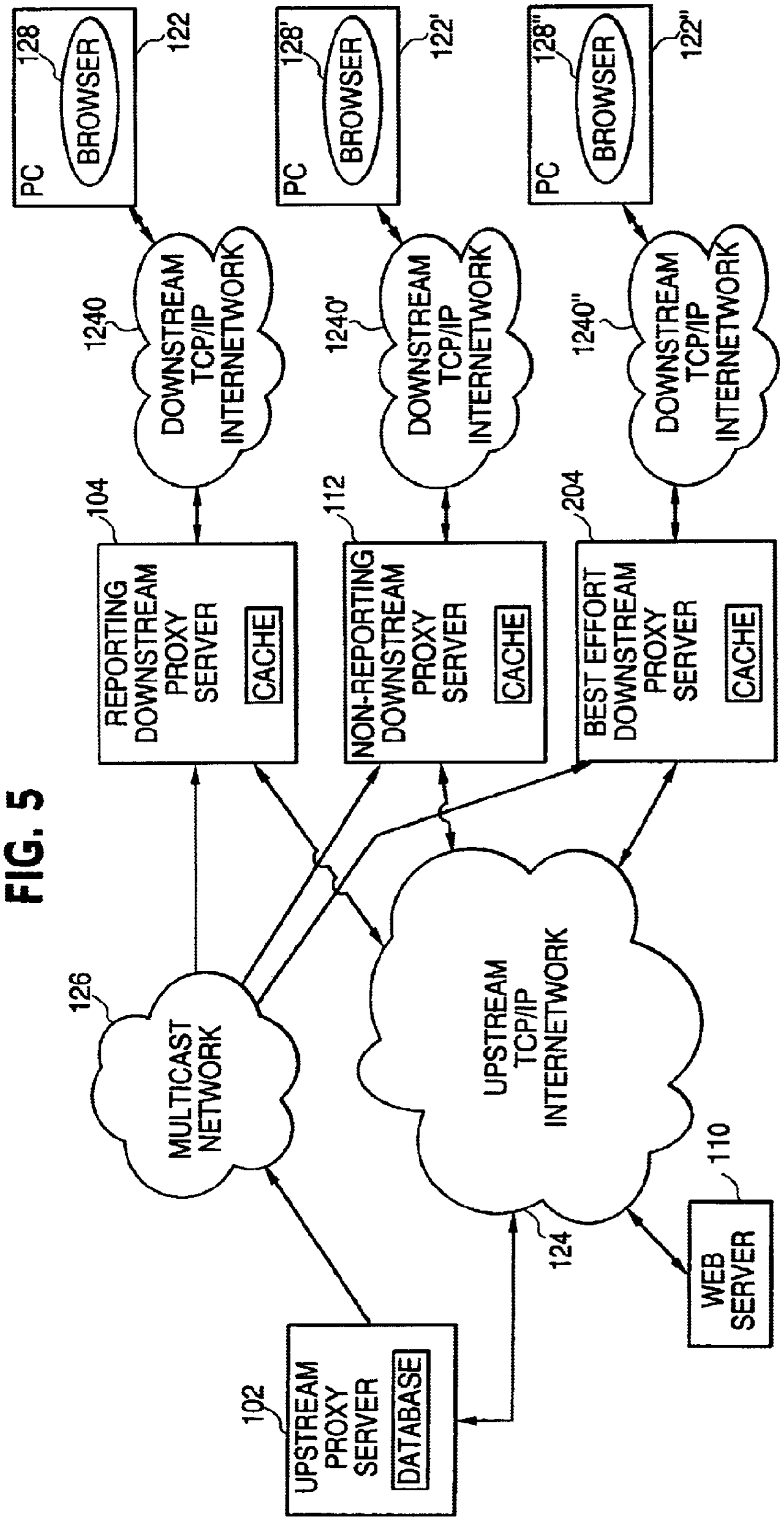

PC
BROWSER
128
122
PC
BROWSER
128'
122'
PC
BROWSER
128"
122"
1240
DOWNSTREAM TCP/IP INTERNETWORK
1240'
DOWNSTREAM TCP/IP INTERNETWORK
1240"
DOWNSTREAM TCP/IP INTERNETWORK
104
REPORTING DOWNSTREAM PROXY SERVER
CACHE
112
NON-REPORTING DOWNSTREAM PROXY SERVER
CACHE
204
BEST EFFORT DOWNSTREAM PROXY SERVER
CACHE
126
MULTICAST NETWORK
UPSTREAM TCP/IP INTERNETWORK
124
110
WEB SERVER
102
UPSTREAM PROXY SERVER
DATABASE

FIG. 5

FIG. 5a

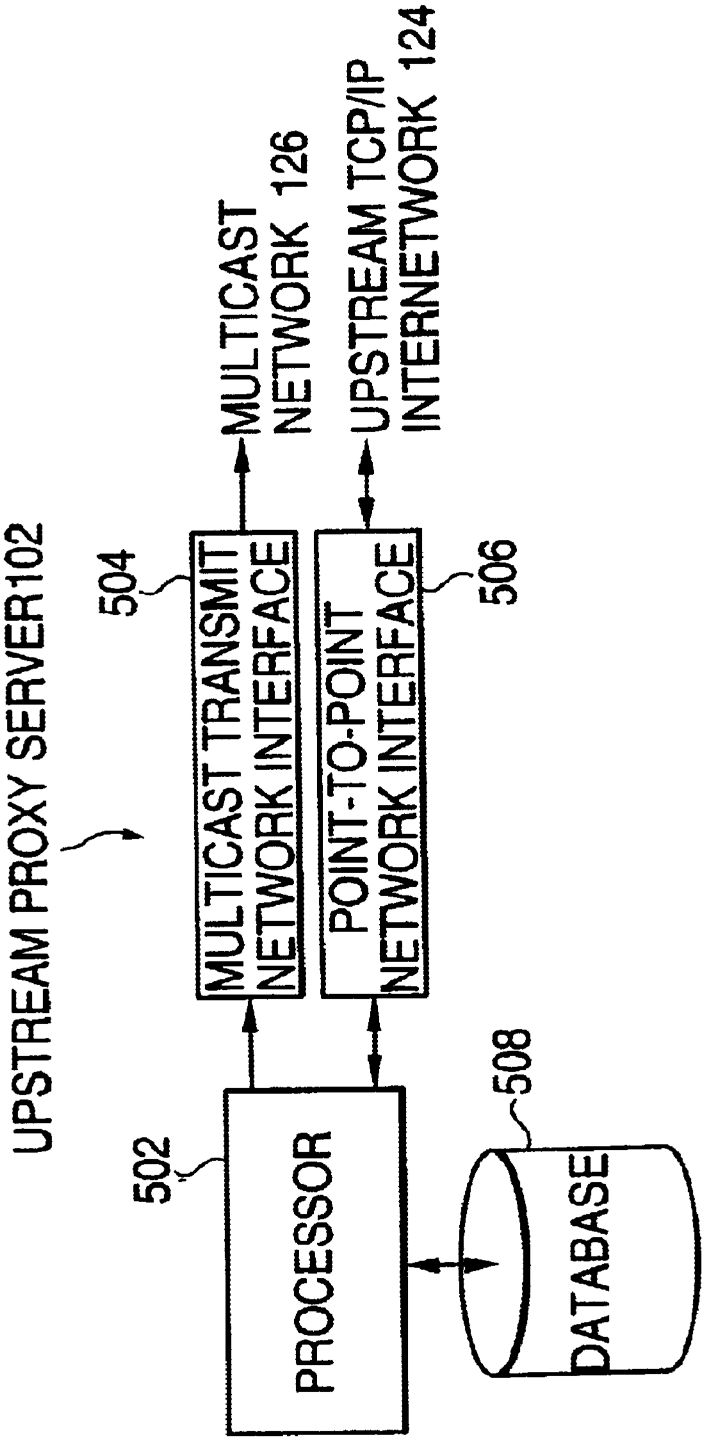

UPSTREAM PROXY SERVER102
504
MULTICAST TRANSMIT NETWORK INTERFACE
MULTICAST NETWORK 126
POINT-TO-POINT NETWORK INTERFACE
506
UPSTREAM TCP/IP INTERNETWORK 124
502
PROCESSOR
508
DATABASE

FIG. 5b

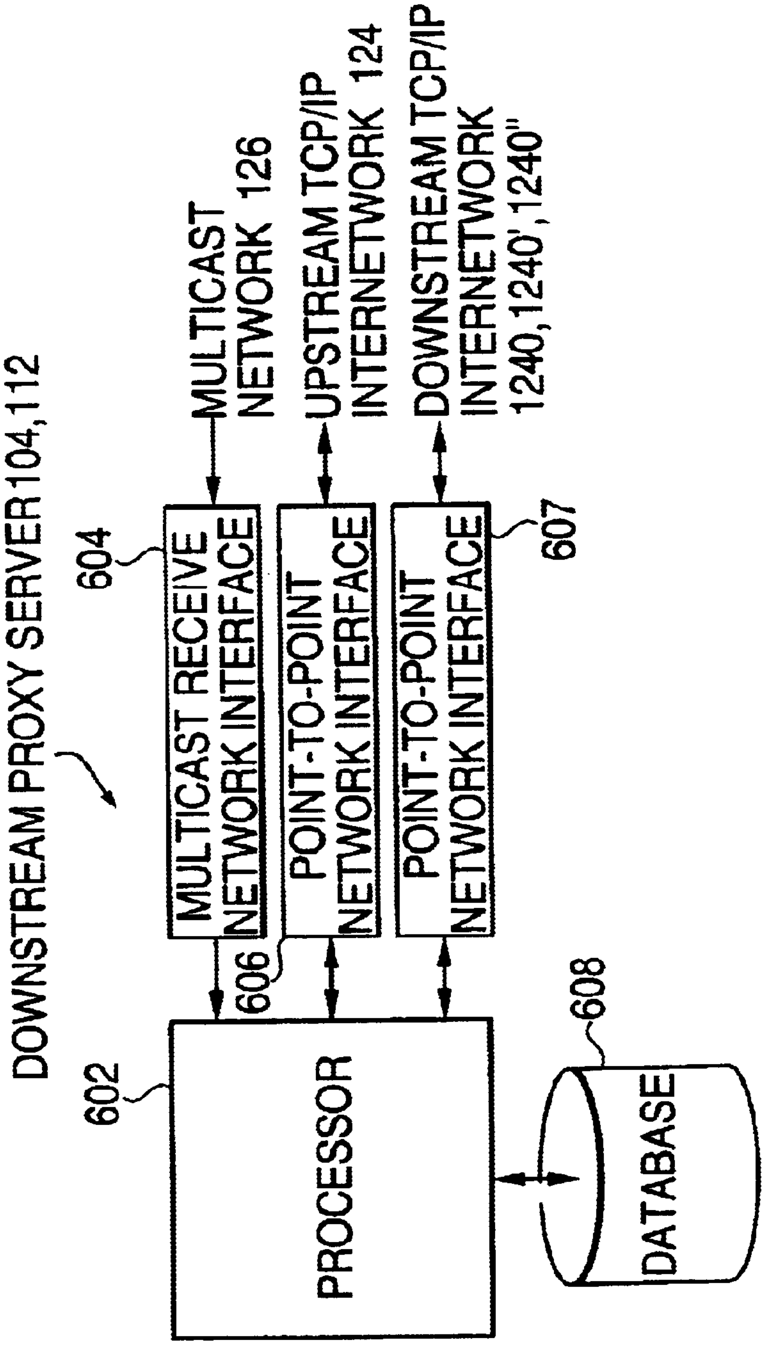
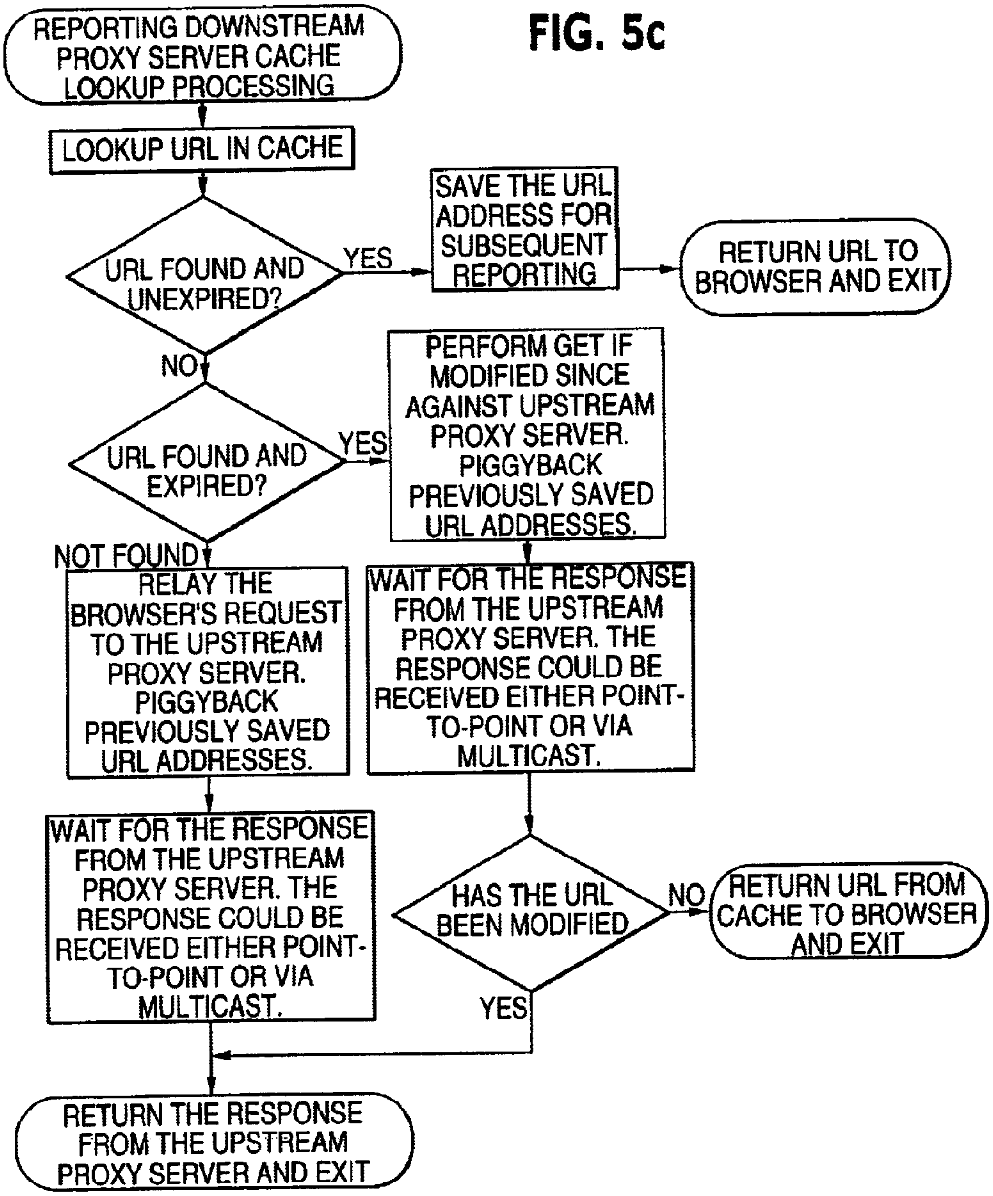

DOWNSTREAM PROXY SERVER 104,112
602
PROCESSOR
604
MULTICAST RECEIVE NETWORK INTERFACE
606
POINT-TO-POINT NETWORK INTERFACE
607
POINT-TO-POINT NETWORK INTERFACE
608
DATABASE
MULTICAST NETWORK 126
UPSTREAM TCP/IP INTERNETWORK 124
DOWNSTREAM TCP/IP INTERNETWORK 1240,1240',1240''

FIG. 5c


REPORTING DOWNSTREAM PROXY SERVER CACHE LOOKUP PROCESSING
LOOKUP URL IN CACHE
URL FOUND AND UNEXPIRED?
YES
SAVE THE URL ADDRESS FOR SUBSEQUENT REPORTING
RETURN URL TO BROWSER AND EXIT
NO
URL FOUND AND EXPIRED?
YES
PERFORM GET IF MODIFIED SINCE AGAINST UPSTREAM PROXY SERVER. PIGGYBACK PREVIOUSLY SAVED URL ADDRESSES.
NOT FOUND
RELAY THE BROWSER'S REQUEST TO THE UPSTREAM PROXY SERVER. PIGGYBACK PREVIOUSLY SAVED URL ADDRESSES.
WAIT FOR THE RESPONSE FROM THE UPSTREAM PROXY SERVER. THE RESPONSE COULD BE RECEIVED EITHER POINT-TO-POINT OR VIA MULTICAST.
WAIT FOR THE RESPONSE FROM THE UPSTREAM PROXY SERVER. THE RESPONSE COULD BE RECEIVED EITHER POINT-TO-POINT OR VIA MULTICAST.
HAS THE URL BEEN MODIFIED
NO
RETURN URL FROM CACHE TO BROWSER AND EXIT
YES
RETURN THE RESPONSE FROM THE UPSTREAM PROXY SERVER AND EXIT

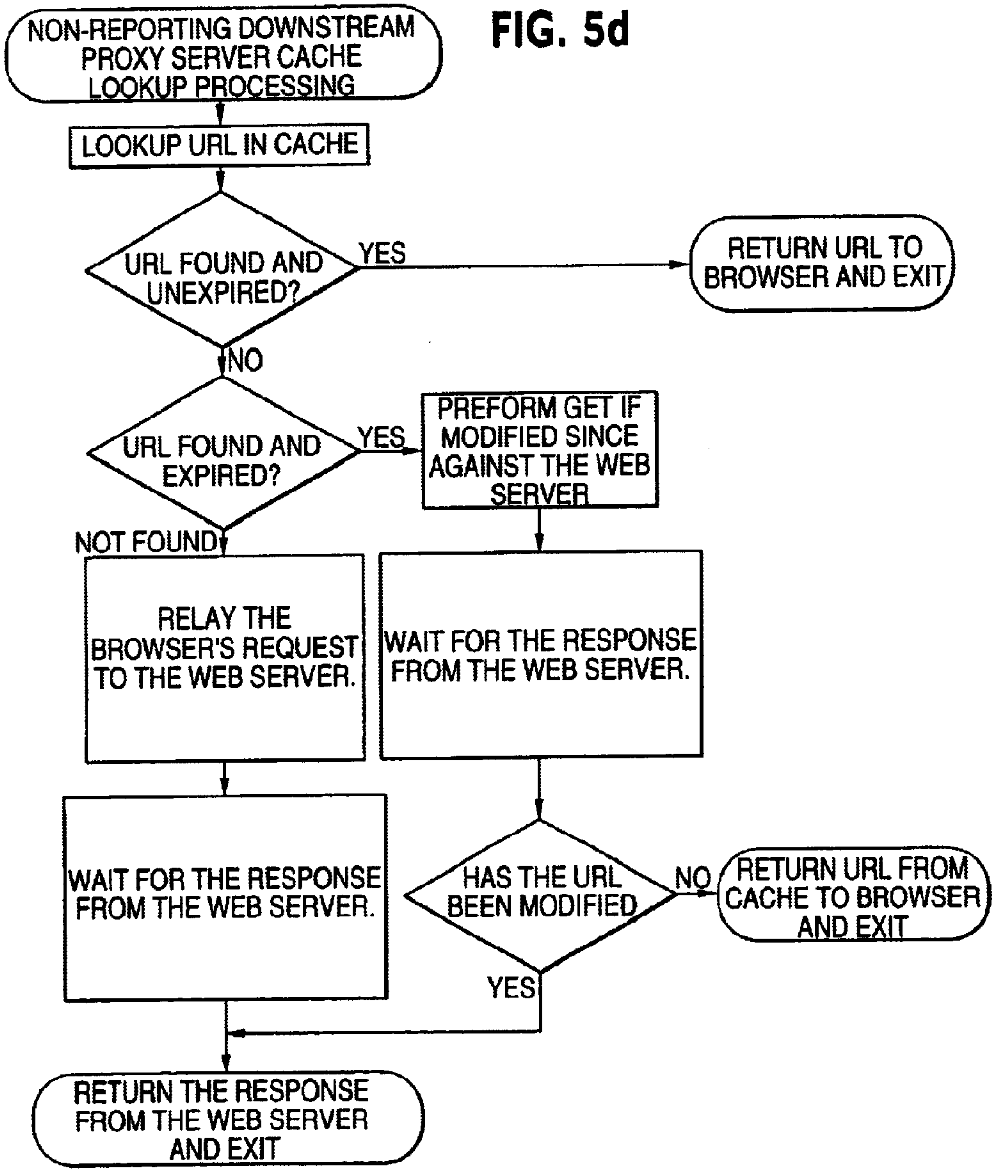
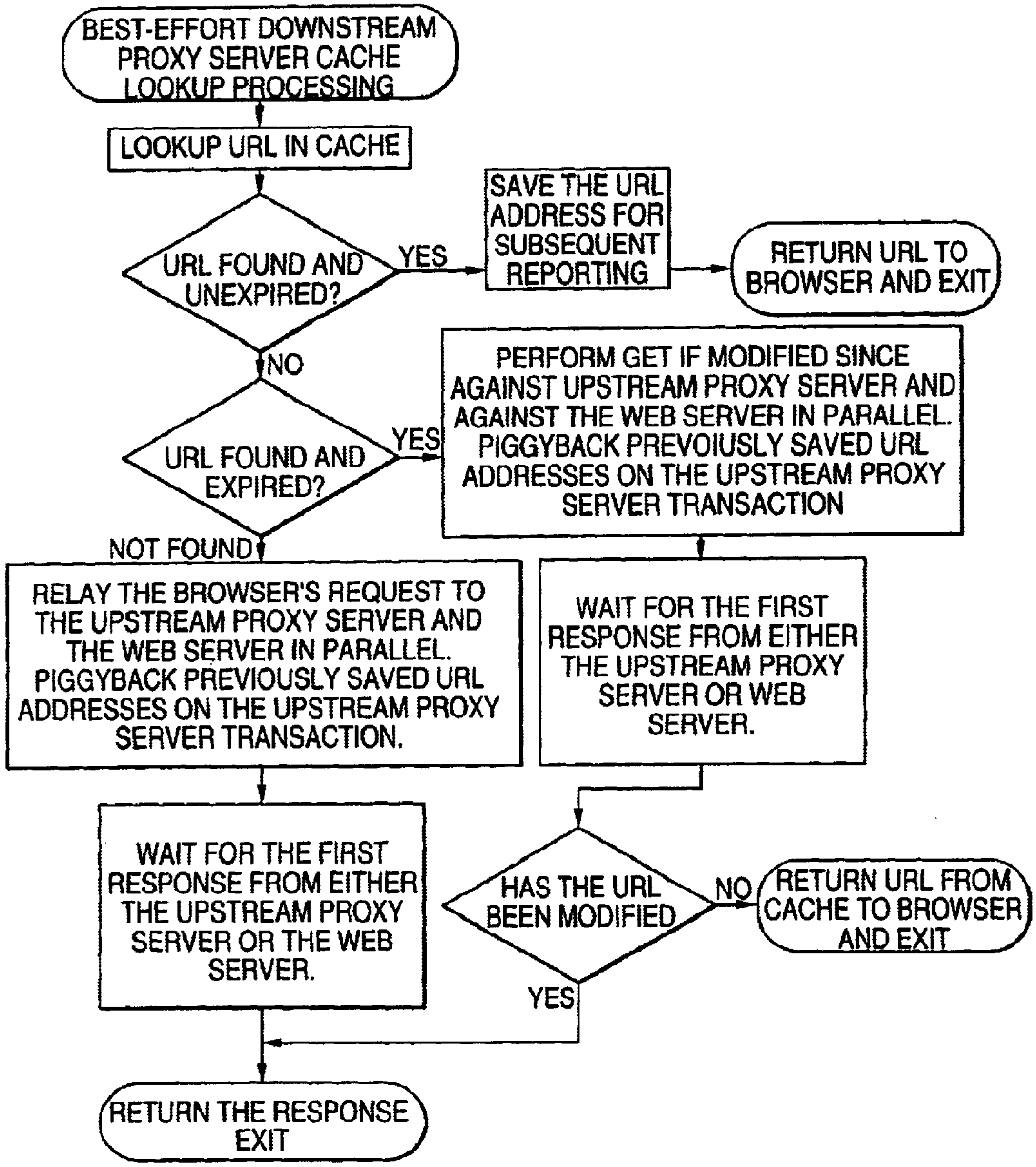

NON-REPORTING DOWNSTREAM PROXY SERVER CACHE LOOKUP PROCESSING
LOOKUP URL IN CACHE
URL FOUND AND UNEXPIRED?
YES
RETURN URL TO BROWSER AND EXIT
NO
URL FOUND AND EXPIRED?
YES
PREFORM GET IF MODIFIED SINCE AGAINST THE WEB SERVER
NOT FOUND
RELAY THE BROWSER'S REQUEST TO THE WEB SERVER.
WAIT FOR THE RESPONSE FROM THE WEB SERVER.
WAIT FOR THE RESPONSE FROM THE WEB SERVER.
HAS THE URL BEEN MODIFIED
NO
RETURN URL FROM CACHE TO BROWSER AND EXIT
YES
RETURN THE RESPONSE FROM THE WEB SERVER AND EXIT

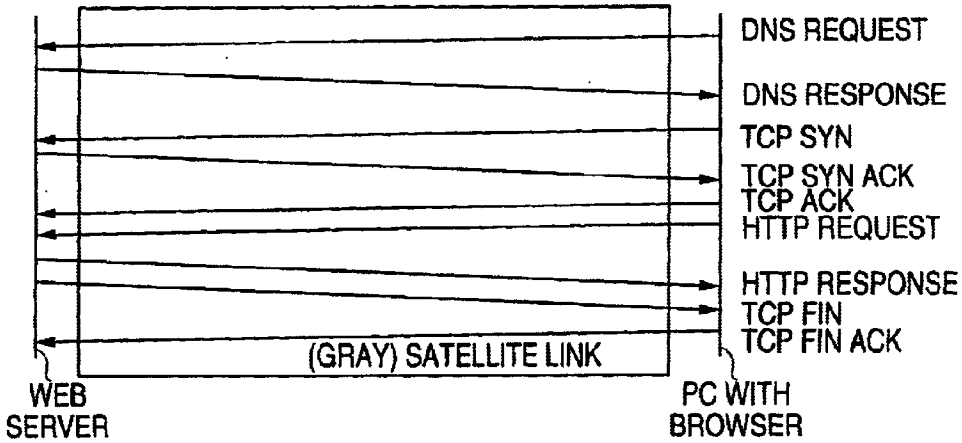
FIG. 5d

FIG. 5e


BEST-EFFORT DOWNSTREAM PROXY SERVER CACHE LOOKUP PROCESSING
LOOKUP URL IN CACHE
URL FOUND AND UNEXPIRED?
YES
SAVE THE URL ADDRESS FOR SUBSEQUENT REPORTING
RETURN URL TO BROWSER AND EXIT
NO
URL FOUND AND EXPIRED?
YES
PERFORM GET IF MODIFIED SINCE AGAINST UPSTREAM PROXY SERVER AND AGAINST THE WEB SERVER IN PARALLEL. PIGGYBACK PREVOIUSLY SAVED URL ADDRESSES ON THE UPSTREAM PROXY SERVER TRANSACTION
NOT FOUND
RELAY THE BROWSER'S REQUEST TO THE UPSTREAM PROXY SERVER AND THE WEB SERVER IN PARALLEL. PIGGYBACK PREVIOUSLY SAVED URL ADDRESSES ON THE UPSTREAM PROXY SERVER TRANSACTION.
WAIT FOR THE FIRST RESPONSE FROM EITHER THE UPSTREAM PROXY SERVER OR WEB SERVER.
WAIT FOR THE FIRST RESPONSE FROM EITHER THE UPSTREAM PROXY SERVER OR THE WEB SERVER.
HAS THE URL BEEN MODIFIED
NO
RETURN URL FROM CACHE TO BROWSER AND EXIT
YES
RETURN THE RESPONSE EXIT

PACKETS TRAVERSING THE SATELLITE LINK FOR A SINGLE HTTP TRANSACTION WITH HTTP 1.0

FIG. 6



PACKETS TRAVERSING THE SATELLITE LINK FOR A SINGLE HTTP TRANSACTION WITH THE PRESENT INVENTION

FIG. 7

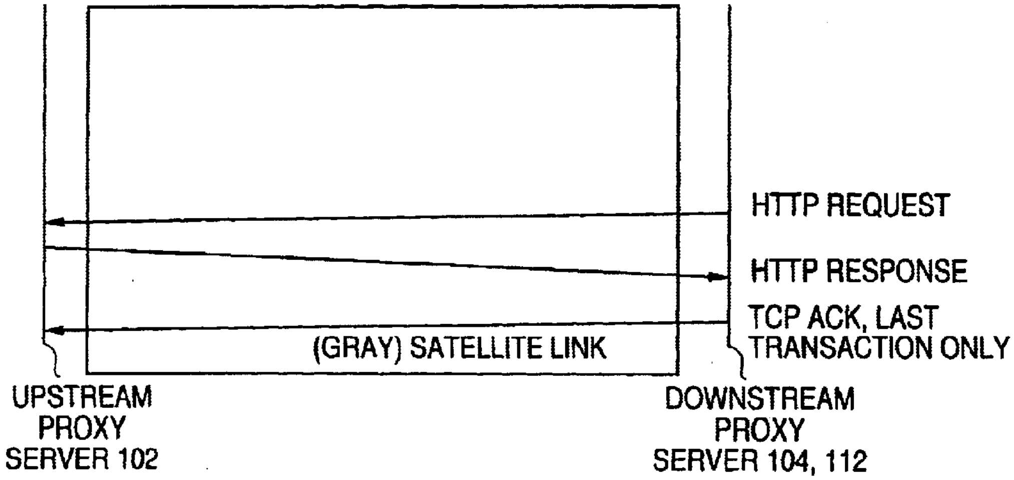

HTTP REQUEST
HTTP RESPONSE
TCP ACK, LAST TRANSACTION ONLY
(GRAY) SATELLITE LINK
UPSTREAM PROXY SERVER 102
DOWNSTREAM PROXY SERVER 104, 112

FIG. 8

```
GET http://www.yahoo.com/HTTP/1.0
PROXY-CONNECTION: KEEP-ALIVE
USER-AGENT:MOZILLA/4.61 [EN] (WinNT;1)
HOST:www.yahoo.com
ACCEPT: IMAGE/GIF, IMAGE/X-XBITMAP,IMAGE/PEG, IMAGE/PJPEG, IMAGE/PNG,1
ACCEPT-ENCODING:GZIP
ACCEPT-LANGUAGE:EN
ACCEPT-CHARSET: ISO-8859-1,*,utf-8
COOKIE: B     31c61vhar6s
```

FIG. 9

```
HTTP/1.0 200 OK

PROXY-CONNECTION: KEEP-ALIVE
CONNECTION: KEEP-ALIVE
CONTENT-LENGTH: 11831
CONTENT-TYPE: text/html
```

SATELLITE MULTICAST PERFORMANCE ENHANCING MULTICAST HTTP PROXY SYSTEM AND METHOD

RELATED APPLICATIONS

This application is based on and claims benefit from provisional application entitled "Satellite Multicast Performance Enhancing Multicast HTTP Proxy System and Method" which was filed on Jun. 10, 1999, and respectively accorded Serial No. 60/138,496.

1. BACKGROUND OF THE INVENTION

1.1 Field of the Invention

The present invention relates generally to the distribution of World Wide Web content over a geosynchronous satellite communications network, and in particular, to satellite communications networks having an outbound high-speed, continuous channel carrying packetized data and either a satellite inbound channel or a terrestrial inbound channel, such as a dialup connection to the Internet.

1.2 Description of related Art

1.2.1 Caching HTTP Proxy Servers

The most popular method for distributing multimedia information is the Internet's World Wide Web. The World Wide Web can be considered to be a set of network accessible information resources. In the World Wide Web, many Web Servers and Web Browsers are connected to the Internet via the TCP/IP protocols and the Web Browsers request web pages and graphics and other multimedia content via the Hypertext Transfer Protocol (HTTP).

The World Wide Web is founded on three basic ideas:

1. A global naming scheme for resources—that is, Uniform Resource Locators (URLs).
2. Protocols for accessing named resources—the most common of which is the Hypertext Transfer Protocol (HTTP).
3. Hypertext—the ability to embed links to other resources which is typically done according to the Hypertext Markup Language (HTML).

Web pages are formatted according to the Hypertext Markup Language (HTML) standard which provides for the display of high-quality text (including control over the location, size, color and font for the text), the display of graphics within the page and the "linking" from one page to another, possibly stored on a different web server. Each HTML document, graphic image, video clip or other individual piece of content is identified, that is, addressed, by an Internet address, referred to as a Uniform Resource Locator (URL). In the context of this invention, a "URL" may refer to an address of an individual piece of web content (HTML document, image, sound-clip, video-clip, etc.) or the individual piece of content addressed by the URL. When a distinction is required, the term "URL address" refers to the URL itself while the terms "URL content" or "URL object" refers to the content addressed by the URL.

A web browser may be configured to either access URLs directly from a web server or from an HTTP proxy server. An HTTP proxy server acts as an intermediary between one or more browsers and many web servers. A web browser requests a URL from the proxy server which in turn "gets" the URL from the addressed web server. An HTTP proxy itself may be configured to either access URLs directly from a web server or from another HTTP proxy server. When a proxy server sends a request to another proxy server the proxy server processing the request is referred to as being upstream (that is, closer to the web server). When a proxy server receives a request from another proxy server, the requesting proxy server is referred to as being downstream, that is, farther from the Web Server.

FIG. **1** illustrates a system in which one of a plurality of browsers accesses a web server via upstream and downstream proxy servers with an HTTP GET command. In particular, a plurality of PCs **12**, each including a browser **14**, output a GET command to web server **16**, in order to access the URL "A". Assuming PC **12** and browser **14** make the first request, the GET command is passed to downstream proxy server **18**. Since this is the first request for URL "A", the downstream proxy server **18** does not have URL "A" in its cache **20**. As a result, the downstream proxy server **18** also issues a GET URL "A" command to upstream proxy server **22**. Since this is also the first request to upstream proxy server **22** for the URL "A", the upstream proxy server **22** also does not have URL "A" in its cache **24**. Therefore, the upstream proxy server **22** issues a GET URL "A" command directly the web server **16**. The web server **16** services this request and provides the upstream proxy server **22** with the desired information, which is then stored in the cache **24**. The upstream proxy server **22** passes the desired information to the downstream proxy server **18**, which also stores the desired information in its cache **20**. Finally, the downstream proxy server **18** passes the desired information to the originating requestor's browser **14** at PC **12**, which also stores the desired information in its cache **21**.

Subsequently, PC **12'**, via its browser **14'**, also desires the information at URL "A". PC **12'** issues a GET URL "A" command to downstream proxy server **18**. At this time, downstream proxy server **18** has the desired information in its cache **20** and provides the information directly to PC **12'** without requesting additional information from either the upstream proxy server **22** or the web server **16**. Similarly, if PC **12"**, via its browser **14"**, also desires the information at URL "A", PC **12"** issues a GET URL "A" command to downstream proxy server **18'**. However, since downstream proxy server **18'** does not have the information for URL "A" stored in its cache **20'**, the downstream proxy server **18'** must access the upstream proxy server **22** and its cache **24**, in order to supply the desired information to PC **12"**. However, the upstream proxy server **22** does not have to access the web server **16**, because the desired information is stored in its cache **24**.

As described above, a caching HTTP proxy server, such as downstream proxy servers **18**, **18'** and upstream proxy server **22** store (cache) some URLs. Normally, a caching proxy server stores the most frequently accessed URLs. When a web server delivers a URL, it may deliver along with the URL an indication of whether the URL should not be cached and an indication of when the URL was last modified. As described in conjunction with FIG. **1**, the URLs stored by a caching proxy server are typically URLs obtained on behalf of a browser or downstream proxy server. A caching HTTP proxy server satisfies a request for a URL, when possible, by returning a stored URL. The HTTP protocol also supports a GET IF MODIFIED SINCE request wherein a web server (or a proxy server) either responds with a status code indicating that the URL has not changed or with the URL content if the URL has changed since the requested date and time.

FIG. **2** illustrates a browser executing a GET IF MODIFIED SINCE command from web server **16**. As illustrated in FIG. **2**, the PC **12**, including browser **14**, has already requested URL "A" once and has URL "A" stored in its cache **21**. PC **12** now wants to know if the information stored at URL "A" has been updated since the time it was last requested. As a result, the browser **14** issues a GET A IF MODIFIED SINCE the last time "A" was obtained. Assuming that URL "A" was obtained at 11:30 a.m. on Jul. 13, 1999, browser **14** issues a GET A IF MODIFIED SINCE Jul. 15, 1999 at 11:30 a.m. request. This request goes to downstream proxy server **18**. If downstream proxy server **18** has received an updated version of URL "A" since Jul. 15, 1999 at 11:30 a.m., downstream proxy server **18** will supply the new URL "A" information to the browser **14**. If not, the downstream proxy server **18** will issues a GET IF MODIFIED SINCE command to upstream proxy server **22**. If upstream proxy server **22** has received an updated URL "A" since Jul. 15, 1999 at 11:30. a.m., upstream proxy server **22** will pass the new URL "A" to the downstream proxy server **18**. If not, the upstream proxy server **22** will issue a GET A IF MODIFIED SINCE command to the web server a. If URL "A" has not changed since Jul. 15, 1999 at 11:30 a.m., web server **16** will issue a NO CHANGE response to the upstream proxy server **22**. In this way, bandwidth and processing time are saved, since if the URL "A" has not been modified since the last request, the entire contents of URL "A" need not be transferred between web browser **14**, downstream proxy server **18**, upstream proxy server **22**, and the web server **16**, only an indication that there has been no change need be exchanged.

Caching proxy servers offer both reduced network utilization and reduced response time when they are able to satisfy requests with cached URLs. Much research has been done attempting to arrive at a near-optimal caching policy, that is, determining when a caching proxy server should store URLs, delete URLs and satisfy requests from the cache both with and without doing a GET IF MODIFIED SINCE request against the web server. Caching proxy servers are available commercially from several companies including Microsoft, Netscape, Network Appliance and Cache Flow.

1.2.2 Satellite Multicast Networks

Typical geosynchronous satellites relay a signal from a single uplink earth station to any number of receivers under the "foot print" of the satellite. FIG. **3** illustrates a typical satellite system **40**. The satellite system **40** includes an uplink earth station **50**, a satellite **52**, and receiving terminals **54**, **54'**, **54''**, **54'''**. The satellite system **40** covers a footprint **56**, which in the example in FIG. **3**, is the continental United States. The footprint **56** typically covers an entire country or continent. Multicast data is data which is addressed to multiple receiving terminals **54**. When the signal is carrying digital, packetized data, a geosynchronous satellite **52** is an excellent mechanism for carrying multicast data as a multicast packet need only be transmitted once to be received by any number of terminals **54**. Such a signal, by carrying both unicast and multicast packets can support both normal point-to-point and multicast applications. Satellite multicast data systems are typically engineered with Forward Error Correcting (FEC) coding in such a way that the system is quasi-error free, that is, under normal weather conditions packets are hardly ever dropped.

The Internet Protocol (IP) is the most commonly used mechanism for carrying multicast data. Satellite networks capable of carrying IP Multicast data include Hughes Network System's Personal Earth Station VSAT system, Hughes Network System's DirecPC™ system as well as other systems by companies such as Gilat, Loral Cyberstar and Media4.

VSAT systems, such as the Personal Earth Station by Hughes Network Systems, use a satellite return channel to support two-way communication, when needed. For World Wide Web access, a terminal using a VSAT system sends HTTP requests to the Internet by means of the VSAT's inbound channel and receive HTTP responses via the outbound satellite channel. Other systems, such as DirecPC's™ Turbo Internet, use dialup modem. (as well as other non-satellite media) to send HTTP requests into the Internet and receive responses either via the outbound satellite channel or via the dialup modem connection. Satellite networks often have a longer latency than many terrestrial networks. For example, the round trip delay on a VSAT is typically 1.5 seconds while the round trip delay of dialup Internet access is typically only 0.4 seconds. This difference in latency is multiplied in the case of typical web browsing in that multiple round trips are required for each web page. This places web browsing via satellite at a distinct disadvantage relative to many terrestrial networks. The present invention provides a major reduction in this disadvantage and as such greatly increases the value of web browsing via satellite.

2. SUMMARY OF THE INVENTION

The present invention is directed to a communication network having an outbound high-speed channel carrying packetized data and either a satellite inbound channel or a terrestrial inbound channel, such as a dial-up connection to the internet. The communication network includes at least one upstream proxy server and at least two reporting downstream proxy servers, where the at least one upstream proxy server is capable of multicasting URLs to the at least two reporting downstream proxy servers. The at least two reporting downstream proxy servers interact with the at least one upstream proxy server to resolve cache misses and the at least one upstream proxy server returns at least one resolution to the cache misses via multicast. The proxy servers included in the communication system may include reporting proxy servers, non-reporting proxy servers, and best effort proxy servers. A reporting downstream proxy server interacts with an upstream proxy server to satisfy a cache miss. A non-reporting downstream proxy server interacts with a web server to satisfy a cache miss. A best effort downstream proxy server requests a cache-miss URL from both the upstream proxy server and the web server.

In one embodiment, the downstream proxy server filters multicast transmissions of URLs and stores the subset of the URLs for subsequent transmission where relative popularity is used to determine whether to store a multicast URL. In one embodiment, the upstream proxy server is capable of multicasting URLs to at least two reporting downstream proxy servers, the upstream proxy server interacts with the two reporting downstream proxy servers to resolve cache misses and the upstream proxy server returns at least one resolution to the cache misses via multicast.

In another embodiment, the downstream reporting proxy server includes a data base and a processor for receiving entries sent by an upstream proxy server, for filtering unpopular entries, keeping popular entries in the database, deleting previously stored entries from the data base, expiring previously stored entries from the data base, or reporting new entries to the upstream proxy server.

As described above, the communication system lowers user response time, lowers network utilization, and reduces the resources required by an HTTP proxy server.

In other embodiments, the present invention is directed to a proxy protocol which performs transaction multiplexing which prevents a single stalled request from stalling other requests, performs homogenized content compression which intelligently compresses HTTP request and response headers and performs request batching so that nearly simultaneously received requests are sent in a single TCP segment, in order to reduce the number of required inbound packets.

3. BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** illustrates a conventional system, including browsers, web servers, upstream and downstream proxy servers, and the execution of a GET COMMAND.

FIG. **2** illustrates a conventional system, including browsers, web servers, upstream and downstream proxy servers, and the execution of a GET IF MODIFIED SINCE COMMAND.

FIG. **3** illustrates a conventional satellite system.

FIG. **4** illustrates a communication system in one embodiment of the preferred invention.

FIG. **5** illustrates a communication system in another embodiment of the present invention.

FIG. **5***a* illustrates an upstream proxy server in one embodiment of the present invention.

FIG. **5***b* illustrates a downstream proxy server in one embodiment of the present invention.

FIG. **5***c* illustrates the cache lookup processing performed by a reporting downstream proxy server in one embodiment of the present invention.

FIG. **5***d* illustrates the cache lookup processing performed by a non-reporting downstream proxy server in one embodiment of the present invention.

FIG. **5***e* illustrates the cache lookup processing performed by a best-effort downstream proxy server in one embodiment of the present invention.

FIG. **6** illustrates the TCP/IP packets which traverse the communication link for a single HTTP transaction without the benefit of the present invention.

FIG. **7** illustrates the TCP/IP packets which traverse the network medium for a single HTTP transaction with the benefit of one embodiment of the present invention.

FIG. **8** illustrates an HTTP request in one embodiment of the present invention.

FIG. **9** illustrates an HTTP response in one embodiment of the present invention.

4. DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

4.1 INTRODUCTION TO THE INVENTION

While there has been some work using satellite multicast to preload an HTTP proxy server cache, the present invention includes several innovations which increase (often dramatically) the utility of such a system when a single user or small number of users access the network through a single satellite multicast receiving proxy. These innovations provide:

1. lower user response time;
2. lower network utilization; and

As depicted by FIG. **4**, in one exemplary embodiment, the present invention includes an upstream, multicasting proxy server **102** and multiple downstream multicast receiving proxy servers **104**, **112**, **204**. The upstream proxy server **102** multicasts web content to the downstream proxy servers **104**, **112**, **204** by means of a one-way capable multicast network such as a geosynchronous satellite broadcast. The one-way capable multicast network includes the satellite uplink **106**, the geosynchronous satellite **108**, and satellite receiver **109**. A subset of the downstream proxy servers **104**, referred to as "reporting proxy servers" interact with the upstream proxy server **102** by means of a two-way point-to-point capable network **120**, examples of which include a dialup access internet network and satellite VSAT systems carrying interactive TCP/IP. Other downstream proxy servers **112** interact with web servers **110** for cache misses without going through the upstream proxy server **102**. These proxy servers **112** are referred to as non-reporting proxy servers. Yet another class of downstream proxy servers **204** interact with both the upstream proxy server **102** and the web server **110** in parallel and passes the HTTP response back to a web browser of PC **122** from whichever responds first. These proxy servers are referred to as best effort proxy servers. In some cases, such as a VSAT system, the multicast network and the two-way point-to-point network may be a single integrated network. In other cases, they may be separate networks.

4.2 PRIOR ART SATELLITE MULTICAST CACHING PROXY SERVER SYSTEMS

There are two classes of known prior art satellite multicast caching proxy server systems.

Multicast push systems, such as, the DirecPC™ TurboWebCast service which is available with DirecPC™ sold by Hughes Network Systems, allow users to subscribe to a set of web channels, where a channel is typically a portion of a web site. The content of the channel is multicast file-transferred to subscribing users and a proxy server on the subscribing user's PC allows the user to access data from the cache offline, without any two-way connection to the Internet.

Large-scale multicast caching systems, such as the system developed by SkyCache multicast content to caching proxies loaded in cable modem head ends and Internet Service Provider points of presence (POPs).

As will be clear from the discussion that follows, the present invention distinguishes from the prior art in several ways, namely:

1. Unlike a multicast push system, the present invention reduces the response time and network utilization experienced by users without requiring any explicit subscription to content by the user and without requiring the preparation and maintenance of channel definitions by the satellite uplink.
2. Unlike large-scale multicast caching systems, the present invention includes novel filtering of multicast URLs to minimize the processing associated with filtering multicast URLs.
3. Unlike large-scale multicast caching systems, the present invention operates correctly and effectively without requiring the continuous operation of a downstream proxy.
4. Unlike large-scale multicast caching systems, the present invention often uses the multicast channel to send URLs in response to a downstream proxy server request thereby reducing the network loading on the point-to-point network connecting the downstream proxy to the upstream proxy. The point-to-point network and the multicast network are often a single, integrated satellite network. When this is the case, multicasting the URL consumes no more network capacity than transmitting it point-to-point while offering the benefit of possibly eliminating future transmissions of the URL by preloading the URL into other receiver's caches.
5. Unlike large-scale multicast caching systems, the present invention allows the downstream proxy server

to automatically cease the processing of multicast traffic when a user is actively using a PC that the downstream proxy server is running on.

6. Unlike large-scale multicast caching systems, the downstream proxy servers pass usage information to the upstream proxy server. The upstream proxy server factors this usage information into its decision whether to multicast URLs.

4.3 SYSTEM OVERVIEW

The term TCP/IP in the context of this invention refers to either the current version of TCP/IP (IP version 4) or the next generations (for example, IP version 6). The basics of TCP/IP internetworking as known by one of ordinary skill in the art, can be found in "Internetworking with TCP/IP Volume 1" by Douglas Comer.

As illustrated in FIG. **5**, the present invention allows web browsers **128**, **128'**, **128"** to access multiple web servers **110**, only one such web server being depicted. The web servers **110** and the upstream proxy server **102** are connected to a TCP/IP internetwork **124** referred to as the upstream internetwork. The upstream proxy server **102** is able to multicast to the downstream proxy servers **104**, **112**, **204** by the multicast network **126**. A subset of the downstream proxy servers **104**, **204** interact with the upstream proxy server **102** by the TCP/IP internetwork **124**. In some cases the upstream intemetwork and the downstream internetwork are actually a single, integrated internetwork. Downstream proxy servers **104**, **112**, **204** are of one of three types which are referred to as follows:

Reporting downstream proxy servers **104** interact with the upstream proxy server **102** exclusively to satisfy a cache miss. Reporting downstream proxy servers **104** also report cache hits to the upstream proxy server **102**. The reporting downstream proxy server **104** is the preferred type of downstream proxy server when the upstream intemetwork **124** naturally routes all traffic from the downstream proxy server **104** through a nodes near the upstream proxy server **102**. This is the case when the downstream proxy server **104** is connected to the Internet via a typical, star topology, two-way VSAT network.

Non-reporting downstream proxy servers **112** interact with the addressed web server **110** to satisfy a cache miss. This interaction with the web server **110** may take place either directly with the web server or by means of an upstream proxy server (not shown in FIG. **5**) which is independent of the multicast capable upstream proxy server **102**. Non-reporting downstream proxy servers **112** do not report cache hits to the upstream proxy server **102**. A non-reporting downstream proxy server **112** is the preferred type of downstream proxy server when reporting downstream proxy servers **204** are operating so as to keep the upstream proxy server's estimation of URL popularity up-to-date and to keep the multicast network filled and when a low-complexity minimal processing and memory resources are desired in a subset of the downstream proxy servers.

Best effort downstream proxy servers **204** request a cache-miss URL from both the upstream proxy server **102** and the addressed web server **110**. The request to web server **110** may optionally be taken either directly to the web server **110** or by means of an upstream proxy server (not shown in FIG. **5**) which is independent of the multicast capable upstream proxy server **102**. The best effort downstream proxy server **204** uses the first complete response from either the upstream proxy server **102** or the web server **110**, the best effort downstream proxy server **204** is referred to as "best effort" in that best effort communications mechanisms are used between the downstream **204** and the upstream proxy server **102** both to request URLs and to report cache hits. The best effort downstream proxy server **204** is the preferred type of downstream proxy server when the upstream internetwork **124** does not naturally route all traffic from the downstream proxy server near the upstream proxy server. There are many examples where this is the case including where the upstream internetwork **124** is accessed by the downstream proxy server via a dialup modem connection.

As illustrated in FIG. **5***a*, the upstream proxy server **102** may include:

1. a processor **502** with RAM memory containing programs and data. As is well known in the art, the processor **502** may be implemented in hardware or software, if in hardware, digitally as discrete or integrated circuits. The processor **502** may also include a single processor or multiple processors, interconnected in parallel and/or serial;
2. a multicast transmit network interface **504** capable of transmitting multicast IP packets via the multicast network **126**;
3. a point-to-point network interface **506** capable of sending and receiving TCP/IP packets via the upstream TCP/IP Intemetwork **124**; and
4. a database **508** accessible by the processor **502** containing the status (and optionally the content) of URLs of interest in the upstream proxy server **102**.

As is well known to those skilled in the art, a single network interface, such as an ethernet interface, with the proper system routing is capable of carrying both multicast and point-to-point traffic and as such, an alternative implementation of the upstream proxy server **102** may utilize a single network interface **504**/**506** to carry both the multicast and point-to-point traffic.

As illustrated in FIG. **5***b*, the downstream proxy server **104**, **112**, **204** may include:

1. a processor **602** with RAM memory containing programs and data. As is well know in the art the processor **602** may in actual practice be a computer containing a single or multiple processors operating in parallel;
2. a multicast receive network interface **604** capable of transmitting multicast IP packets via the multicast network **126**;
3. a point-to-point network interface **606** capable of sending and receiving TCP/IP packets via the upstream TCP/IP Internetwork **124**;
4. a point-to-point network interface **607** capable of sending and receiving TCP/IP packets via the downstream TCP/IP Internetwork **1240**, **1240'**, **1240"**; and
5. a database **608** accessible by the processor **602** containing a domain name cache containing entries for the domain names or IP addresses of web-servers **110**, **110'**, **110"** popularly accessed by clients of the downstream proxy server **104**, **112** and a URL cache containing URLs and associated content which can be provided to clients should they request them.

As is well known to those skilled in the art, a single network interface, such as an ethernet interface, with the proper system routing is capable of carrying both multicast and point-to-point traffic and as such, an alternative implementation of the downstream proxy server **104**, **112** may utilize a single network interface to carry both the multicast and upstream TCP/IP Internetwork and downstream TCP/IP Internetwork traffic. Other alternatives include the use of two network interfaces with one of the network interfaces carrying the traffic of two of the network interfaces enumerated above.

The upstream proxy server **102** determines which URLs to multicast and multicasts the URLs and information summarizing URL status. An HTTP URL begins with the string "http://" followed by either the domain name or the IP address of the web server which can serve the URL. The upstream proxy server **102** multicasts URLs in such a way to facilitate the filtering of URLs by web server domain name or IP address. When the upstream proxy server **102** multicasts a URL, it multicasts the URL, the HTTP response header associated with the URL and expiration information for the URL.

Downstream proxy servers **104, 112, 204** maintain a domain name cache from recently accessed URLs. The cache has a maximum size and when a new item is inserted into a full cache, an older, less frequently accessed domain name must be removed to make room for the new item. The downstream proxy servers **104, 112, 204** maintain the relative popularity of each domain name in the cache where popularity is defined by the frequency of HTTP requests to the site. The downstream proxy servers **104, 112, 204** filter out all multicast URLs (and URL status) except those from the most popular entries in the domain name cache. URLs which pass the filter are candidates for being cached.

When a browser **128** requests a URL from the downstream proxy servers **104, 112, 204** the downstream proxy servers **104, 112, 204** update the popularity of that domain name's cache entry adding a new entry for the URL's domain name if not already present. The downstream proxy server **104, 112** then looks up the URL in the downstream proxy server's URL cache. What happens after this depends on whether the URL is found in the cache, whether the URL has expired and whether the downstream proxy server is a reporting **104**, non-reporting **112**, or best-effort **204** server.

The downstream proxy server **104, 112, 204** directly returns the URL to the browser when the URL is found in the cache and the URL has not expired. A reporting proxy server **104** or a best effort proxy server **204** saves the URL address for subsequent reporting to the upstream proxy server **102**. When found and unexpired, both user response time and network utilization are reduced.

The downstream proxy server **104, 112, 204** performs a GET IF MODIFIED SINCE operation against the upstream proxy server **102** when the URL is found and is expired. The downstream proxy server **104, 112, 204** thus checks to make sure the content is up to date.

When a cache lookup finds the URL in the cache and the URL is expired, the processing that takes place depends on the type of downstream proxy server.

A reporting downstream proxy server **104** performs a GET IF MODIFIED SINCE operation against the upstream proxy server **102** when the URL is found and is expired. The reporting downstream proxy server **104** piggybacks any saved URL addresses on the GET IF MODIFIED SINCE request.

A non-reporting downstream proxy server **112** performs a GET IF MODIFIED SINCE operation against the web server **110** when the URL is found and is expired.

A best effort downstream proxy server **204** performs a GET IF MODIFIED SINCE operation against both the upstream proxy server **102** and against the web server **110** in parallel when the URL is found and is expired.

Network utilization and response time are reduced by the present invention in the case of a downstream proxy cache hit of an expired URL provided the GET IF MODIFIED SINCE transaction indicates that the URL has not changed. This is because the actual URL content need not traverse the upstream internetwork **124**.

When the URL is not found in the downstream proxy server's cache, the processing that takes place depends on the type of downstream proxy server.

When the cache lookup fails, a reporting downstream proxy server **104** relays the web browser's **128** GET or GET IT MODIFIED SINCE transaction to the upstream proxy server **102** piggybacking any unreported saved URL addresses. As will be discussed later, response time is often reduced even for this case as an HTTP transaction is performed across the proxy-to-proxy link is typically faster than a browser to web server HTTP transaction.

When the cache lookup fails, a non-reporting downstream proxy server **112** relays the web browser's **128'** GET or GET IT MODIFIED SINCE transaction to the web server.

When the cache lookup fails, a best effort downstream proxy server **204** relays the web browser's **128"** GET or GET IF MODIFIED SINCE transaction to both the upstream proxy server **102** and to the web server. As will be discussed later, response time may be reduced even for this case if the upstream proxy server responds to this transaction and the response arrives sooner than the web server's response.

FIGS. **5***c*, **5***d*, and **5***e* illustrate the processing flow for performing cache lookup for the reporting downstream proxy server **104**, non-reporting downstream proxy server **112**, and best-effort downstream proxy server **204**, respectively.

The upstream proxy server **102** keeps a database of URLs. In some implementations the upstream proxy server **102** is a caching server. When the upstream proxy server **102** is a caching server the URL database may either be integrated with the cache or operate independently of the cache. When the upstream proxy server **102** receives a request for a URL, in some cases it produces a full HTTP response, either from its cache or by interacting with the web server **110** or interacting with a yet further upstream proxy server (not shown in FIG. **5**). The upstream proxy server **102** then looks up the URL in its database, updates its entry (or creates the entry if one does not already exist), and determines, based on various criteria discussed later, whether to respond at all and whether to multicast the response. The upstream proxy server **102** returns a point-to-point HTTP response to the reporting downstream proxy server **104** regardless of whether a multicast response is being sent. When a multicast response is being sent, the point-to-point HTTP response signals the reporting downstream proxy server **104** to receive the response via multicast. The upstream proxy server **102** only returns a point-to-point response to a best effort downstream proxy server **204** when the response indicates that the URL is not expired and not modified. Responses containing URL content, when sent to a best effort downstream proxy server **204**, are sent only via multicast.

The downstream proxy servers **104, 112, 204** use their domain name caches to efficiently filter and process URLs which have been multicast. The downstream proxy servers **104, 112, 204** discard multicast URLs for domain names not present in the domain name cache. The mechanism for multicasting URLs and for discarding URLs based on domain name is optimized, as will be described in detail later, to reduce the processing required by the downstream proxy servers **104, 112, 204**. The downstream proxy servers **104, 112, 204** receive and process multicast URLs for a subset of the domain names in the cache. This subset includes the domain names for which the downstream proxy has an outstanding request to the upstream proxy server **102** and the domain names which the caching policy determines as being most likely to have URLs which will be worth storing in the cache.

A reporting or best effort downstream proxy's **104, 204** domain name cache is organized so that when the proxy server **104, 204** has an HTTP request outstanding to the upstream proxy server **102** that the domain name (or IP address) from the requested URL address is locked in the domain name cache. This ensures that when a response is multicast, the response passes the downstream proxy server's **104, 204** filter and will be processed by the downstream proxy server **104, 204**.

When the downstream proxy server **104, 112, 204** receives a multicast URL or URL status update, it submits the URL to its URL caching policy. The caching policy decides whether to store the multicast URL, delete a previously stored URL or expire a previously stored URL. In this way, the downstream proxy server **104, 112, 204** builds up its URL cache with URLs which may be accessed at a later time.

In many systems, such as two-way star-network VSAT systems, both multicast responses and point-to-point responses are carried on a single outbound channel. In such systems, multicasting a response has the benefit of potentially preloading the cache of many receivers while taking no more outbound bandwidth than a point-to-point response. Preloading a cache with a URL reduces network utilization and response time when a successful lookup for that URL occurs at a later time. Overall, the multicasting of responses has the twin benefits of reducing network utilization and response time.

5. PROXY TO PROXY PROTOCOL

5.1 INTRODUCTION

The World Wide Web's use of HTML and the HTTP 1.0 protocol, the version currently in use by almost all browsers, is both inefficient and has very slow response time when operating over satellite networks. This is because HTTP requires a separate TCP connection for each transaction. Multiplying the inefficiency is HTML's mechanism for creating frames and embedding images which requires a separate HTTP transaction for every frame and URL. This particularly affects VSAT networks which have a relatively long round trip delay (1.5 sec) and are cost-sensitive to the number of inbound packets.

FIG. **6** illustrates the packets which traverse the satellite link for a single HTTP transaction and the typical response time. FIG. **7** illustrates the packets which traverse the satellite link (or other network medium) for a single HTTP transaction in the present invention when a reporting downstream proxy server **104** performs the transaction against the upstream proxy server **102**. Table 1 illustrates the cumulative effect of this on a HTML page, like www.cnn.com, containing 30 URLs in terms of the total number of inbound packets and the total delay for accessing such a web page. Table 1 also shows the beneficial effects of acknowledgement reduction as described in U.S. Pat. No. 5,985,725 to Dillon entitled "Method and Apparatus for Requesting and Retrieving Information for a Source Computer Using Terrestrial and Satellite Interface" issued Nov. 30, 1999, without which the packet count would be much higher.

As can be seen in Table 1, the response time for a 30 URL web page goes from 16.5 seconds without the present invention to 7.5 seconds with the present invention. This is better than a 2 to 1 reduction in response time. As also can be seen from Table 1, the number of inbound packets per 30 URL web page goes from 121 to 30, a better than 4 to 1 reduction.

TABLE 1

Response Time And Inbound Packets For A 30 URL Web Page Over VSAT

	Inbound Packets	Response Time (sec)	Description
Time For Individual Operation With HTTP 1.0			
A	1	1.5	Domain Name Lookup
B	4	3	One HTTP Get (assuming no web server delay)
Time For An Entire Web Page With HTTP 1.0 (assume 8 browser connections)			
A + B	5	4.5	The HTML URL (first)
B	32	3	First 8 embedded images
B	32	3	Second 8 embedded images
B	32	3	Third 8 embedded images
B	20	3	Last 5 embedded images
	121	16.5	Total For 30 URL Web Page
Time For Individual Operation With Present Invention			
C	0	0	Domain Name Lookup (no satellite round trip, performed by upstream proxy)
D	1	1.5	One HTTP Get (no connection establishment)
E	2	1.5	Last HTTP Get (ack for last data)
Time For An Entire Web Page With Present Invention			
C + D	1	1.5	The HTML URL (first)
D	8	1.5	First 8 embedded images
D	8	1.5	Second 8 embedded images
D	8	1.5	Third 8 embedded images
E	4	1.5	Last 5 embedded images
	29	7.5	Total For 30 URL Web Page

With HTTP 1.1, the proposed enhancement to HTTP 1.0, which improves the response time and networking efficiency of HTTP 1.0, the present invention provides even greater improvements. The present invention provides better compression than HTTP 1.1 and, unlike HTTP 1.1, does not allow a single slow or stalled HTTP request to slow down other requests.

5.2 PROXY-TO-PROXY (P2P) PROTOCOL OVERVIEW AND BENEFITS

The present invention replaces the HTTP protocol often used between upstream **102** and reporting downstream **104** proxy servers with a protocol optimized for this role referred to as the Proxy-To-Proxy (P2P) protocol. The P2P protocol carries HTTP transactions between the downstream **104** and upstream **102** proxy servers.

The P2P protocol carries HTTP 1.0 and 1.1 request and response headers and content where the request and response headers include extensions to support multicast cache pre-loading.

Apart from these multicast header extensions, the P2P protocol improves over HTTP transport in the following ways:

Transaction Multiplexing—improves over separate connection for each transaction (HTTP 1.0) and pipelining

(HTTP 1.1) by preventing a single stalled request from stalling other requests. This is particularly beneficial when the downstream proxy server **104** is supporting simultaneous requests from multiple browsers **128**, **128'**, **128"**.

Homogenized Content Compression—improves over HTTP 1.1 content compression by intelligently compressing HTTP request and response headers and by allowing compression streams for common data to extend over multiple URLs. This increases the overall compression ratio. HTTP 1.1 does not compress request and response headers. This is particularly important when the inbound channel is a shared wireless medium such as a VSAT inroute or some other wireless medium. It effectively allows many more subscribers to share the available inbound bandwidth.

Request Batching—batches HTTP requests which arrive at nearly the same time so that the requests get sent over the satellite in a single TCP segment, thereby reducing the number of inbound packets.

5.3 TRANSACTION MULTIPLEXING

The P2P protocol rides on top of a general purpose protocol, the TCP Transaction Multiplexing Protocol (TTMP). TTMP allows multiple transactions, in this case HTTP transactions, to be multiplexed onto one TCP connection.

The downstream proxy server **104** initiates and maintains a TCP connection to the upstream proxy server **102** as needed to carry HTTP transactions. The TCP connection could be set up and kept connected as long as the downstream proxy server **104** is running and connected to the downstream internetwork **124**. It could also be set up when the first transaction is required and torn down after the connection has been idle for some period.

An HTTP transaction begins with a request header, optionally followed by request content which is sent from the downstream proxy server **104** to the upstream proxy server **102**. This is referred to as the transaction request. An HTTP transaction concludes with a response header, optionally followed by response content. This is referred to as the transaction response.

The downstream proxy server **104** maintains a transaction ID sequence number which it increments with each transaction. The downstream proxy server **104** breaks the transaction request into one or more blocks, creates a TTMP header for each block, and sends the blocks with a TTMP header to the upstream proxy server **102**. The upstream proxy server **102** similarly breaks a transaction response into blocks and sends the blocks with a TTMP header to the downstream proxy server **104**. The TTMP header contains the information necessary for the upstream proxy server **102** to reassemble a complete transaction command and to return the matching transaction response. The TTMP header contains:

The transaction ID—the transaction sequence number must rollover less frequently than the maximum number of supported outstanding transactions.

Block Length—allows a proxy server **102, 104, 112, 204** to determine the beginning and ending of each block. As is well known by those skilled in the art, byte stuffing and other techniques can be used, rather than length fields, to identify the beginning and ending of blocks of data.

Last Indication—allows the proxy server **102, 104, 112, 204** to determine when the end of a transaction response has been received.

Abort Indication—allows the proxy server **102, 104, 112, 204** to abort a transaction when the transaction request or response cannot be completed.

Compression Information—defines how to decompress the block as explained in more detail in Section 3.4 below.

By breaking transaction requests into blocks and allowing the blocks from different transactions to be interleaved, the P2P protocol of the present invention benefits from allowing a single TCP connection to simultaneously carry multiple HTTP requests without allowing a single stalled (partially received) transaction request or response to block other transactions. The P2P protocol also allows transaction response information to be relayed back to the downstream proxy server **104** in the order it is provided by the various web servers **110**, again preventing a stalled or slow web server from delaying URLs from other web servers **110**.

The use of a single HTTP connection, rather than the multiple connections used with HTTP 1.0 and optionally with HTTP 1.1 reduces the number of TCP acknowledgements sent over the inbound medium. Reduction in the number of TCP acknowledgements significantly reduces the use of inbound networking resources which, as said earlier, is very important when the inbound is a shared medium such as a VSAT or other wireless medium. This reduction of acknowledgements is more significant when techniques, such as those described in U.S. Pat. No. 5,985,725 to Dillon entitled "Method and Apparatus for Requesting and Retrieving Information for a Source Computer Using Terrestrial and Satellite Interface" issued Nov. 30, 1999, minimize the number of TCP acknowledgements per second per TCP connection.

For example, the Hughes Network Systems DirecPC™ Enterprise Edition networking product reduces the number of TCP acknowledgements per connection sent over the satellite link to two per second regardless of the amount of traffic flowing on that connection. Without the present invention, a web browser **110** might utilize eight HTTP 1.0 connections in parallel across the satellite link. With the present invention, only a single connection is run. As a result, the present invention reduces the inroute acknowledgement traffic eight-fold. If multiple web browsers **110** are operating in parallel, this reduction in inbound acknowledgement traffic is further increased.

5.4 HOMOGENIZED CONTENT COMPRESSION

HTTP 1.1 defines a single algorithm for compressing URL content and each URL's content is individually compressed. The P2P protocol achieves a higher compression ratio than HTTP 1.1 as follows:

Does not restart a compression algorithm with each data item being compressed.

Uses algorithms optimized to the type of data being compressed.

Compresses HTTP request and response headers.

5.4.1 Introduction to Lossless Compression

Lossless compression algorithms can in general be classified into two broad types: statistics-based and dictionary-based. Statistics-based compression algorithms exploit the probability distribution of the data to encode the data efficiently. Two well-known algorithms of this type are Huffman coding and arithmetic coding. The process of statistics-based coding can be split into two parts: a modeler that estimates the probability distribution of data, and an encoder that uses the probability distribution to compress the data. According to information theory, it is possible to construct an optimal code that asymptotically comes arbitrarily close to the entropy rate of the data. Huffman coding can achieve asymptotic optimality by blocking symbols into large groups, however this is computationally infeasible. Moreover, Huffman coding is not suitable for handling adaptive statistics of data. On the other hand, arithmetic coding overcomes these drawbacks of Huffman coding and achieves asymptotic optimality without sacrificing computational simplicity. The essential drawbacks of statistics-based coding are their slower speed (compared to dictionary-based algorithms), and the inaccuracies in statistical modeling of data. Regarding the latter issue, accurate modeling of data requires sophisticated statistical techniques, which in turn require large amount of training data, which is often unavailable.

On the other hand, dictionary-based compression algorithms achieve compression by replacing a string of symbols with indices from a dictionary. The dictionary is a list of strings that are expected to occur frequently in the data. Such a dictionary could either be a static pre-defined dictionary, or an adaptive dictionary that is built and updated as data is compressed. Dictionary-based compression algorithms usually require less computational resources than statistics-based techniques. Lempel-Ziv (LZ) type compression is a general class of adaptive dictionary-based lossless data compression algorithms. LZ-type compression algorithms are based upon two distinct type of approaches—LZ77 and LZ78. An LZ77-type algorithm adaptively builds a dictionary only at the transmitter end of the lossless connection. Compression is achieved by sending a pointer along with size of match of matching phrases occurring in the already compressed portion of the data stream. The receiver end of the connection does not need to maintain a dictionary, thereby minimizing memory requirements, and can decode the compressed data very quickly. LZSS is a commonly used lossless data compression algorithm based on LZ77.

Commonly used LZ78-type lossless data compression algorithms, for example LZW and the algorithm used in the ISO V.42bis specification, use a dictionary (or other data structure) which is built up on both ends of a lossless connection, such as a TCP connection, as data is transferred across the link. Compression is achieved by sending a reference into the dictionary in place of an uncompressed string of bytes. The references are constructed to be smaller than the original string, but sufficient to restore the original string together with the dictionary. These algorithms automatically tune the dictionary as data is transferred so that the dictionary is well prepared to provide high compression should data similar to earlier previously transferred data be submitted for compression.

The term compression stream refers to a compressor and decompressor each with their own dictionary at opposite ends of a lossless connection. For LZ78-type algorithms, encoding is faster than LZ77-type algorithms, however the decoders are slower and have considerably higher memory requirements.

These algorithms are much more efficient when they are processing relatively large amounts of similar data. For example, one HTML page is very similar to subsequent HTML pages, especially when the pages come from the same web server **110**. Maximum compression is not obtained immediately after a dictionary is initialized, as it has not been tuned to compress the data at hand.

5.4.2 P2P Use of Lossless Data Compression

The data passed across the P2P protocol can be categorized into the following groups of data:

1. HTTP Request And Response Headers
2. HTML—sent from the upstream proxy **102** to the downstream proxy **104**, **112**.
3. Precompressed data—entity bodies (URL content) such as JPEG and GIF images which are known to be precompressed and do not benefit from further compression attempts.
4. Other—other entity body (URL content) data.

The P2P protocol of the present invention maintains a separate compression stream in each direction for each category of data. This ensures, for example, that downstream HTML data is sent through a compression stream whose dictionary is well tuned for processing HTML providing a higher compression ratio than could be expected from individually compressing each HTML page.

The P2P protocol of the present invention uses compression algorithms for each category of data which are efficient for their category of data. For example, P2P uses:

1. HTTP Requests and Response Headers each are fully text, with plenty of standard keywords. This motivates the use of a dictionary-based algorithm whose dictionary is constructed using the frequently occurring standard keywords. Such a scheme is further improved by combining the static dictionary approach with the power of adaptive dictionary approach of LZ-type algorithm. The static dictionary compresses the standard phrases, while LZ compresses those phrases that are not standard but repeat in the data.
2. HTML data is fully text and warrants the use of a data compression algorithm optimized for text. LZW works well with text and may be used for HTML data when using a more highly optimized algorithm is not convenient.
3. No compression is used for precompressed data to avoid wasting CPU attempting to compress data, which cannot be further compressed.
4. Other data is compressed with a general purpose compression algorithm such as LZW.

As is well known to those skilled in the art, an HTTP request includes an HTTP request header optionally followed by a message body. An HTTP request header further includes a series of one-line, ASCII strings with each string referred to as an HTTP request header field. The end of an HTTP request header is delimited by an empty line. FIG. **8** illustrates a typical HTTP request header.

As is also well known to those in the skilled in the art, an HTTP response includes an HTTP response header optionally followed by a message body. An HTTP response header further includes a series of one-line, ASCII strings with each string referred to as an HTTP response header field. The end of an HTTP response header is delimited by an empty line. FIG. **9** illustrates a typically HTTP response header. The first line of an HTTP response includes the status code, a 3 digit decimal number, which summarizes the type of response being provided. In the example in FIG. **9**, a 200 response code is being provided which means that the response was "OK" and that the message body contains the requested URL.

5.5 HTTP 1.1 EXTENSIONS TO SUPPORT MULTICAST CACHE PRELOADING

The present invention includes an addition to a HTTP request used to report cache hit usage reporting and an addition to an HTTP response to direct a reporting downstream proxy server **104** to expect the HTTP response via multicast rather than via P2P. The use of the additions will be explained in detail below.

5.5.1 HITREP HTTP Request Extension

By adding a HITREP header field to an HTTP request a reporting downstream proxy 104 or a best effort downstream proxy server 204 may report to the upstream proxy 102 cache hits which occurred on unexpired URLs. A HITREP attribute is formatted as follows:

HITREP=comma separated list of URLs.

The upstream proxy 102 removes this attribute prior to forwarding the HTTP request to the web server 110. It uses the information to update its hit database entries of the contained URLs in a way which increases the likelihood that updates to these URLs will be multicast.

5.5.2 Mcast Status Code

The upstream proxy 102 directs a reporting downstream proxy server 104 to expect the URL to be sent via multicast by returning a Mcast Status Code. In one embodiment, the Mcast Status Code has a value of 360.

5.6 Best Effort Downstream Proxy Server To Upstream Proxy Server Transactions

The best effort downstream proxy server 204, for efficiency, uses the User Datagram Protocol (UDP) to transmit HTTP GET and GET IF MODIFIED SINCE requests to the upstream proxy server 102. This is done by placing the HTTP request header into the UDP payload. In order to piggyback cache hit reports on an HTTP request, the HTTP request header may contain a HITREP field as described earlier.

The use of UDP is very efficient as the overhead of establishing, maintaining and clearing TCP connections is not incurred. It is "best effort" in that lost UDP packets are not recovered. Even with lost packets, the upstream proxy server 102 obtains a very representative view of the URLs of interest to the best effort downstream proxy servers.

6. MULTICAST PROXY TO PROXY PROTOCOL

6.1 URL TRANSPORT

The Advanced Television Enhancement Forum (ATVEF) has published their 1.0 specification containing a description of the Unidirectional Hypertext Transport Protocol (UHTTP). UHTTP defines a method for multicasting URLs. The present invention uses UHTTP with extensions to transport URLs through the multicast network 126 and uses special multicast addressing to minimize the CPU time expended discarding data not of interest.

6.2 MULTICAST ADDRESSING

The multicast system of the present invention is improved to support a single web browser 128 or a small number of browsers 128. When a small number of users (10 or less) is involved, the history of previously visited sites is a strong predictor of future accesses. The present invention leverages this insight to dramatically reduce the processing required to filter multicast URLs. Large scale multicast proxy systems do not leverage this insight as it does not apply significantly when many users are accessing a proxy.

The upstream proxy server 102, when deciding whether to multicast a URL, also classifies the URL as being either of general or specific appeal. General URLs are sufficiently popular to be multicast to all the downstream proxies 104, 112, 204 regardless of the downstream proxy's previous history of sites visited. An example, where this might be applicable would be NASA's Jet Propulsion Laboratory's when the comet hit Jupiter. Many users which had never visited JPL's website would go to that site.

The upstream proxy server 102 multicasts general URLs on a single IP multicast address.

To minimize processing of specific URLs, the upstream proxy server 102 spreads the transmission of specific URLs over a large group of multicast addresses. In a preferred embodiment, the upstream proxy server 102 takes the domain of a specific URL and performs a hash function (or any other technique known to one of ordinary skill in the art) to select the multicast address on which the URL is to be multicast. Alternatively, in another embodiment, the upstream proxy server 102 takes the IP address corresponding to the source of the URL and performs a hash function (or any other technique known to one of ordinary skill in the art) on the IP address to select the multicast address on which the URL is to be multicast. The use of hash functions is well understood within the computer. science community and is introduced in "Data Structures and Algorithms in C++," Adam Drozdek, PWS Publishing Co., Sections 10–10.1, 1996. The downstream proxy servers 104, 112, 204 utilize the same hash function to determine, from their domain name cache, the set of multicast addresses to open. This mechanism allows a downstream proxy server 104, 112, 204 to use the destination multicast IP address to filter out most of the specific URLs. For downstream proxy servers 104, 112, 204 that have hardware multicast address filtering this effectively eliminates almost all the CPU time spent on filtering specific URLs. Even downstream proxy servers 104, 112, 204 without hardware address filtering can more efficiently filter URLs based on destination address rather than digging into a packet and filtering based the domain name. Filtering based on domain names is based on string comparison operations and is inherently slower than filtering based on unsigned integer comparisons as is the case with hash functions.

Table 2 illustrates how the use of hash function addressing reduces downstream proxy server 104, 112, 204 processing by efficiently limiting the traffic which must be processed.

Hash Function Addressing Filter Effectiveness

50,000	MulticastAddresses Assigned To Carry Specific URLs
100	Domain Names In The Domain Name Cache
1/500	Fraction Of Specific URL Multicast Traffic Passing The Multicast Address Filter

6.3 HTTP 1.1 EXTENSIONS TO SUPPORT MULTICAST PRELOADING

The present invention includes extensions to the HTTP response header to guide a downstream proxy server 104, 112, 204 processing of multicast URLs.

6.3.1 URL Popularity

The upstream proxy server 102 adds a URLPopularity field to an HTTP response. This field identifies the relative popularity of the URL to other URLs which are being multicast. The URL Popularity field holds an 8 digit unsigned hexadecimal number. The field contains the AgedAccessNumber further discussed below.

6.3.2 Mcast Expiration

The upstream proxy server 102 also adds a McastExpiration header field to an HTTP response. This field contains, like an Expires field, an HTTP-date field. It may also contain 0 which means consider the URL expired. The downstream proxy server 104, 112, 204 uses this field to determine whether to validate its URL cache entry by making a GET IF MODIFIED SINCE request.

7. UPSTREAM PROXY MULTICAST POLICY

It is expected that the upstream proxy server's 102 multicast policy will be improved over time. The implementation of the present invention allows this policy to be enhanced without disrupting the operation of the receiver. The policy described here provides a clear mechanism for reducing overall outbound network utilization.

7.1 URL ADDRESS DATABASE

The upstream proxy server **102**, as described earlier, receives from reporting and best effort downstream proxy servers **104**, **204** requests for URLs and cache hit reports. The upstream proxy server **102** uses these requests and usage reports to maintain a URL address database. This database contains URL address entries, each of which contains:

URL Address—the URL address itself or a message digest of the address (see the discussion of message digests below).

AgedAccessCounter—a 32-bit unsigned counter which is increased with every request for the-URL and with each usage report for the URL and which is reduced to age out stale entries.

ExpirationTime—holds the GMT time when this URL expires.

The upstream proxy server **102** maintains the AgedAccessCount such that it is an indicator of its URL's popularity, that is, frequency of access by downstream proxy servers **104**, **112**. The upstream proxy server **102**, upon receiving a request or a usage report for a URL, looks up the URL in its database, if found, increases its AgedAccessCount, for example, by 1000. The upstream proxy server **102** creates an entry with the AgedAccessCount initialized to a default initial value (e.g. 1000) if the URL was not found. Periodically, (e.g. hourly), the upstream proxy server **102** reduces each database entry's AgedAccessCount by a configurable amount.(e.g. 10%).

7.2 MESSAGE DIGESTS

As is well known to a practitioner skilled in the art, a message digest (or digest) is a relatively short (e.g. 64 bits), fixed length string of bits which is a function of a variable length string of bits. This function has the property that the message digest of different variable length strings will almost always have different digests. "Almost always" means, in this case, a very low probability (e.g. 1 in 2^{60} or one in 10^{18}). Some message digest functions also have the useful property in cryptographic systems that it is difficult to create a string whose message digest is identical to the message digest of another string. This property is not required for this invention. Message digests are introduced in "Applied Cryptography" by Bruce Schneier. The present invention utilizes message digests to determine when two URL addresses are identical (by checking whether their digests are identical).

7.3 MULTICAST NETWORK UTILIZATION

The upstream proxy server **102** is configured with a maximum multicast outbound bit rate, for example, 6 Mbits/sec. The upstream proxy server **102** manages its multicast transmissions to not exceed this maximum rate. In the preferred embodiment, the upstream proxy server **102** maintains twenty byte counters, one for each tenth of a second. It moves round robin from one counter to the next every tenth of a second. When it multicasts a packet, it adds the size of the packet to the counter. Thus, the upstream proxy server **102** can calculate the average throughput over the last two seconds. From this average throughput, the upstream proxy server **102** can calculate the overall multicast utilization, that is, the average throughput divided by the maximum multicast outbound bit rate.

The upstream proxy server **102** is also configured with a maximum general multicast outbound bit rate, for example, 1 Mbit/sec. The upstream proxy server **102** manages its multicast transmission of general URLs to not exceed this maximum rate. This is done in a fashion similar to overall multicast transmission, the upstream proxy server **102** can calculate its general multicast utilization.

7.4 HTTP RESPONSE EXPIRATION FIELD

The upstream proxy server **102** receives HTTP responses either from a web server **110** or a yet further upstream proxy server (not shown in FIG. **5**). Prior to multicasting a URL, the upstream proxy server **102** must ensure that there is an appropriate expiration field in the HTTP response header.

The policy for calculating the expiration is as follows:

1. If any cookies were present in the HTTP request, the response may be specific to the requesting browser and the upstream proxy server **102** sets the Expiration field to 0, indicating already expired. As is well known to those skilled in the art, a cookie is a data item which is provided by a web server to a browser in an HTTP response and is returned to the web server in subsequent HTTP requests. It is typically used to allow the web server to identify the requests which are coming from a single user. A cookie HTTP request header field is shown in the typical request illustrated in FIG. **8**.
2. Otherwise, if the expiration field already exists, the upstream proxy server **102** leaves it untouched.
3. Otherwise, the upstream proxy server **102** sets the expiration field based on MIME type. The upstream proxy server **102** is configured with a table giving the expiration duration for various MIME types and a default expiration for all other MIME types. The upstream proxy server **102** takes the current GMT time and adds the appropriate expiration duration to calculate the expiration time. This allows HTML (which is more likely to change) to expire sooner than images (gif and jpg) which are less likely to change).

7.5 MULTICAST DECISION

The upstream proxy server **102** receives HTTP responses either from a web server **110** or a yet further upstream proxy server (not shown in FIG. **5**). The upstream proxy server **102** examines the HTTP response header to determine the cachability of the URL.

If it is uncachable and the request came from a reporting downstream proxy server, the upstream proxy server **102** returns the response to the downstream proxy server **104** by its point-to-point connection. If it is cacheable, the upstream proxy server **102** looks up its URL in the URL address database and determines whether to return a response to the downstream proxy server **104**, **204** and how that response is returned. A response must be returned to a reporting downstream proxy server **104**. The response may be sent either via multicast or via point-to-point connection. A response, if necessary, is returned to a best. effort downstream proxy server **204**, via multicast. Multicast responses may either be sent on the general or on a specific multicast address. The preferred embodiment of the present invention may utilize the following algorithm to determine how, if at all, the upstream proxy server returns a response.

1. Determine whether the URL is qualified to be general multicast, multicast on the general address if qualified.

2. If not, determine whether the URL is qualified for specific multicast, multicast on a specific address if qualified.
3. Otherwise, send it point-to-point if the request came from a reporting downstream proxy server. Send no response otherwise.

7.5.1 General Multicast Decision

The general multicast decision is based on whether the URL content is included in the response, the popularity of the URL and the general multicast utilization where as the utilization goes up, the popularity of the URL also must go up for the URL to be qualified to be transmitted.

The URL content is not included in a "not modified" response to a GET IF MODIFIED SINCE request. Such a response is only qualified to be multicast when the corresponding entry in the URL address database is "expired" and the response itself is not expired. A "qualified" response with no URL content is worth multicasting as it may be used to update the expiration time of the corresponding entry in the cache of the downstream proxy servers.

The upstream proxy server **102** is configured with a general multicast decision table. This table contains a set of entries, each entry containing a general multicast utilization threshold and a minimum AgedAccessCount. A URL is qualified for general multicast transmission if there is any entry in the table where the general multicast utilization is less than the general multicast utilization threshold and AgedAccessCount exceeds the minimum AgedAccessCount. To avoid overloading the general multicast maximum bit rate, the table always contains an entry for 100% utilization which requires an infinitely high AgedAccessCount and the table allows no other entries with a utilization of 100% or higher.

7.5.2 Specific Multicast Decision

The specific multicast decision is based on the popularity of the URL and the overall multicast utilization where, as the utilization goes up, the popularity of the URL also must go up for the URL to be qualified to be transmitted.

The upstream proxy server **102** is configured with a specific multicast decision table. This table contains a set of entries, each entry containing a overall multicast utilization threshold and a minimum AgedAccessCount. A URL is qualified for specific multicast transmission if there is any entry in the table where the specific multicast utilization is less than the specific multicast utilization threshold and AgedAccessCount exceeds the minimum AgedAccessCount. To avoid overloading the overall multicast maximum bit rate, the table always contains an entry for **100**% utilization which requires an infinitely high AgedAccessCount and the table allows no other entries with a utilization of 100% or higher.

7.5.3 Preferred Site Access To Multicast

It may be desirable to give certain web sites preferred access to the multicast channel. The present invention accommodates this by allowing "preferred" reporting and best effort downstream proxy servers **104, 204** to be configured and to configure the upstream proxy server **102** to preferentially multicast requests from "preferred" reporting and best effort downstream proxy servers **104, 204**. The upstream proxy server **102** multicasts all responses to requests coming from a "preferred" downstream proxy server **104, 204**, giving the site priority to the multicast bandwidth and queuing the responses until bandwidth is available. A web crawler program, such a Teleport Pro by Tennyson Maxwell (www.tenmax.com) is then programmed to periodically crawl such a preferred web site. This results in the preferred web site's content being periodically multicast. A preferred downstream proxy server **104, 112** can be configured to either have its responses multicast either as general multicasts (for sites which are very much preferred) or as specific multicasts (for sites which are preferred).

8. DOWNSTREAM PROXY CACHING POLICY

8.1 CACHING POLICY OVERVIEW

It is expected that the downstream proxy server's **104, 112, 204** cache policy will also be improved over time. The implementation of the present invention allows this policy to be enhanced without changing the interface to the upstream proxy server **102**. The policy described here provides a clear mechanism for reducing overall outbound network utilization.

The cache policy of the preferred embodiment of the present invention is optimized for small-scale operation where the downstream proxy server **104, 112, 204** is supporting either a single browser **128** or a small number of browsers and where these browsers have their own caches. The policy supplements the benefits of a browser cache with most of the benefits of the large-scale cache while consuming a fraction of a large-scale cache's resources.

The cache policy includes four separate operations:

1. determining which multicast addresses to open;
2. determining what to do with URLs received on those addresses;
3. aging cache entries in a fashion identical to the upstream proxy server's URL address database entry aging; and
4. cache lookup.

8.2 MULTICAST ADDRESS POLICY

8.2.1 Multicast Reception Modes

The multicast receiver for the downstream proxy server **104, 112, 204** operates in one of two modes:

active—the downstream proxy server **104, 112, 204** opens multicast addresses and actively processes the received URLs on those addresses.

inactive—the downstream proxy server **104, 112, 204** disables multicast reception from the upstream proxy server **102**. In the inactive state the downstream proxy server **104, 112, 204** minimizes its use of resources by, for example, closing the cache and freeing its RAM memory.

For downstream proxy server **104, 112, 204** operating on a general purpose personal computer, the multicast receiver for the downstream proxy server **104, 112, 204** may be configured to switch between the active and inactive states to minimize the proxy server's interfering with user-directed processing. The downstream proxy server **104, 112, 204** utilizes an activity monitor which monitors user input (key clicks and mouse clicks) to determine when it should reduce resource utilization. The downstream proxy server **104, 112, 204** also monitors for proxy cache lookups to determine when it should go active.

Upon boot up, the multicast receiver is inactive. After a certain amount of time with no user interaction and no proxy cache lookups (e.g. 10 minutes), the downstream proxy server **104, 112, 204** sets the multicast receiver active. The downstream proxy server **104, 112, 204** sets the multicast receiver active immediately upon needing to perform a cache lookup The downstream proxy server **104, 112, 204** sets the multicast receiver inactive whenever user activity is detected and the cache has not had any lookups for a configurable period of time (e.g. 5 minutes).

For downstream proxy servers **104, 112, 204** running on systems with adequate CPU resources to simultaneously handle URL reception and other applications, the user may configure the downstream proxy server **104, 112, 204** to set the multicast receiver to stay active regardless of user activity.

8.2.2 Multicast Address Selection

The downstream proxy server **104, 112, 204** is configured to open a configurable number of multicast addresses, for example, 150 addresses. When the downstream proxy server sets the multicast receiver active, the downstream proxy server **104, 112, 204** always opens the general multicast address and the specific multicast addresses for the web sites for which it has outstanding requests to the upstream proxy server **102**. It opens the specific addresses corresponding to the most popular domain names in its domain name cache with the remaining address slots. Reporting downstream proxy servers and best effort downstream proxy servers give priority to the domain names of URLs for which they have outstanding IITTP requests open to the upstream proxy server and close specific addresses as needed to make room for the addresses associated with those URLs. The downstream proxy server **104, 112, 204** thus has access to the multicast of the web sites it is most likely, based on past history, to access.

8.2.3 Multicast URL Reception Processing

A downstream proxy server **104, 112, 204** may receive via multicast either a complete HTTP response with the URL content or "not modified" HTTP response header without URL content with an updated McastExpiration field.

The downstream proxy server **104, 112, 204** examines the URL popularity field of a complete HTTP response and removes URLs from the cache until there is room for the URL just received. The downstream proxy server removes URLs beginning with those with the lowest AgedAccess-Counter values. The downstream proxy server **104, 112, 204** discards a received URL when there are insufficient URLs whose AgedAccessCounter fields are lower than the URL popularity field of the URL just received to make room for the URL just received. When storing the URL just received in the cache, the downstream proxy server **104, 112, 204** copies the URL popularity field into the cache entry's AgedAccessCount.

Upon receiving a "not modified" HTTP response header without URL content, the downstream proxy server **104, 112, 204** looks up the corresponding URL in its cache. If found, the downstream proxy server updates the cache entry's AgedAccessCounter value with the URL popularity field and updates the entry's expiration field with the response header's McastExpiration field's value.

After updating the cache, a reporting or best effort downstream proxy server **104, 204** checks whether an HTTP request is outstanding to either the webserver or upstream proxy server for the received URL. If so and the URL is now in the cache, the downstream proxy server responds to the requesting browser with the cache entry. If a "not-modified" URL response was received and a request for the URL is outstanding and there was no cache entry the downstream proxy server **104, 204** returns the "not-modified" HTTP response to the browser.

8.2.4 URL Cache Aging

The downstream proxy server **104, 112, 204** ages URLs the same way the upstream proxy **102** ages URLs.

8.2.5 Cache Lookup

When the downstream proxy server **104, 112, 204** looks up a URL in the cache and the URL has not expired, the downstream proxy server **104, 112, 204** returns the URL from the cache to the browser **128**. When the URL has expired, the downstream proxy server **104, 112, 204** sends a GET IF MODIFIED SINCE transaction against the upstream proxy server **102** and/or the web server **110** as is appropriate for the category of proxy server receiving the request.

9. CONCLUSION

As set forth above, the present invention offers many significant innovations over prior satellite systems multicast systems by offering lower response time and lower network utilization while limiting the resources required within the satellite receiver and associated equipment needed to achieve these benefits.

Although several embodiments of the present invention have been described above, there are of course numerous other variations that would be apparent to one of ordinary skill in the art. For example, one or more of the downstream proxy servers **104, 112, 204** could reside with the browser **128, 128', 128"** on a single personal computer **122, 122', 122"**. Additionally, one or more of the downstream proxy servers **104, 112, 204** could reside with the browser **128, 128', 128"** on a television set-top box. Further, one or more of the downstream proxy servers **104, 112, 204** residing with the browser **128, 128', 128"** on a single personal computer **122**, may also have a downstream TCP/IP internetwork connection to other browsers which may or may not be operating on personal computers. Also, one or more of the downstream proxy servers **104, 112, 204**, residing with a browser **128, 128', 128"** on a television set-top box, may also have a downstream TCP/IP internetwork connection to other browsers which may or may not be operating on personal computers. Also, the multicast network **126** need not be based on geosynchronous satellite technology but could be based on any of a number of other multicast technologies including wireless terrestrial broadcast systems.

The invention being thus described, it will be obvious that the same may be varied in many ways. Such variations are not to be regarded as a departure from the spirit and scope of the invention, and all such modifications as would be obvious to one skilled in the art are intended to be included within the scope of the following claims.

What is claimed is:

1. A communications system comprising:

at least one upstream proxy server; and

at least two reporting downstream proxy servers;

said at least one upstream proxy server capable of multicasting URLs to said at least two reporting downstream proxy servers;

said at least two reporting downstream proxy servers interacting with said at least one upstream proxy server to resolve cache misses;

wherein said at least one upstream proxy server returns at least one resolution to the cache misses via multicast,

where said at least two reporting downstream proxy servers utilize a relative frequency that a source web server of a multicast URL has had items requested by clients of at least one of said at least two reporting downstream proxy servers to determine whether to store a multicast URL.

2. The communication system of claim **1**, where said at least one upstream proxy server returns at least one response to the cache misses via point-to-point transmission.

3. The communication system of claim **2**, where said at least two reporting downstream proxy servers send cache hit information to said at least one upstream proxy server.

4. The communication system of claim **3**, where said at least two reporting downstream proxy servers piggyback cache hit information on HTTP request headers being sent to said at least one upstream proxy server.

5. The communication system of claim **4**, where said at least one upstream proxy server uses relative frequency accesses of the URL including both cache misses and cache hits as reported to the upstream server to help determine whether content of a cache miss is returned via multicast or point-to-point transmission.

6. The communication system of claim **2**, where said at least one upstream proxy server uses popularity, where popularity is based on a relative frequency of access of a URL, to determine whether the URL is returned via multicast or point-to-point transmission.

7. The communication system of claim **5**, where said at least one upstream proxy server also uses a loading of the multicast channel in combination with the popularity to determine whether the URL is returned via multicast or point-to-point transmission.

8. The communication system of claim **2**, where said at least one upstream proxy server maps a domain name of a source of a URL to a multicast address to determine the multicast address to be used to carry the URL.

9. The communication system of claim **8**, wherein said at least one upstream proxy server maps domain names to multicast addresses using of a hash function.

10. The communication system of claim **8**, wherein URLs with a relatively high popularity are carried on a multicast address dedicated to carrying URLs of general interest.

11. The communication system of claim **1**, wherein additional reporting downstream proxy servers which are not currently interacting with said at least one upstream proxy server filter multicast cache resolutions from said at least one upstream proxy server and store a subset of cacheable items for subsequent retrievals upon request, by a client.

12. The communication system of claim **1**, wherein additional non-reporting downstream proxy servers which do not report to said at least one upstream proxy server filter multicast cache resolutions from said at least one upstream proxy server and store a subset of cacheable items for subsequent retrieval, upon request, by a client.

13. A communication system comprising:

at least one multicast capable upstream proxy server; and

at least two best-effort downstream proxy servers;

said at least one multicast capable upstream proxy server capable of multicasting URLs to said at least two best-effort downstream proxy servers where said at least two best-effort downstream proxy servers interact with said at least one upstream proxy server and either a web-server directly or at least one non-multicast capable proxy server to resolve cache misses;

wherein said at least one multicast capable upstream proxy server returns at least one resolution to the cache misses via multicast; and

wherein said at least two best-effort downstream proxy servers relay responses from said at least one multicast capable proxy server to a client when the responses arrive prior to a response from the web server or said at least one non-multicast capable upstream proxy server.

14. The communication system of claim **13**, where said at least two best-effort downstream proxy servers use a best-effort communication mechanism to send cache miss resolution requests to said at least one multicast capable upstream proxy server.

15. The communication system of claim **13**, where said at least two best-effort downstream proxy servers send cache hit information to said at least one multicast capable upstream proxy server.

16. The communication system of claim **15**, where said at least two best-effort downstream proxy servers piggyback cache hit information on HTTP request headers sent to said at least one multicast capable upstream proxy server.

17. An upstream proxy server capable of multicasting URLs to at least two reporting downstream proxy servers;

said upstream proxy server interacting with said at least two reporting downstream proxy servers to resolve cache misses;

wherein said upstream proxy server returns at least one resolution to the cache misses via multicast,

where said at least one upstream proxy server returns at least one response to the cache misses via point-to-point transmission,

where the upstream proxy server is able to receive cache hit information from at least one downstream proxy server,

where the upstream proxy server uses a relative frequency of cache misses and cache hits to an individual server to determine whether content of a cache miss is returned via multicast or point-to-point transmission.

18. The upstream proxy server of claim **17**, where the upstream proxy server is able to receive cache hit information from at least one downstream proxy server piggybacked on an HTTP request from said downstream proxy server.

19. The upstream proxy server of claim **17**, where the upstream proxy server uses popularity, where popularity is based on a relative frequency of access of a URL, to determine whether the URL is returned via multicast or point-to-point transmission.

20. The upstream proxy server of claim **19**, where the upstream proxy server also uses a loading of the multicast channel in combination with the popularity to determine whether the URL is returned via multicast or point-to-point transmission.

* * * * *

US006128653A

United States Patent [19]

del Val et al.

[11] **Patent Number: 6,128,653**

[45] **Date of Patent: Oct. 3, 2000**

[54] **METHOD AND APPARATUS FOR COMMUNICATION MEDIA COMMANDS AND MEDIA DATA USING THE HTTP PROTOCOL**

[75] Inventors: **David del Val**, Mountain View; **Anders Edgar Klemets**, Sunnyvale, both of Calif.

[73] Assignee: **Microsoft Corporation**, Redmond, Wash.

[21] Appl. No.: **08/822,156**

[22] Filed: **Mar. 17, 1997**

[51] **Int. Cl.**^{7} **G06F 13/14**

[52] **U.S. Cl.** **709/219**; 709/203; 455/4.2

[58] **Field of Search** 345/327–329; 395/200.47–200.49, 200.33; 455/4.2; 348/7, 12, 13; 709/217–219, 203

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,931,950	6/1990	Isle et al.	364/513
5,119,474	6/1992	Beitel et al.	395/154
5,274,758	12/1993	Beitel et al.	395/154
5,341,474	8/1994	Gelman et al.	395/200
5,414,455	5/1995	Hooper et al.	348/7
5,455,910	10/1995	Johnson et al.	395/650
5,533,021	7/1996	Branstad et al.	370/60.1
5,537,408	7/1996	Branstad et al.	370/79
5,623,690	4/1997	Palmer et al.	395/806
5,721,827	2/1998	Logan et al.	395/200.47
5,732,219	3/1998	Blumer et al.	395/200.47
5,754,772	5/1998	Leaf	395/200.33
5,793,966	8/1998	Amstein et al.	395/200.33
5,796,393	8/1998	MacNaughton et al.	345/329
5,796,566	8/1998	Sharma et al.	361/86

FOREIGN PATENT DOCUMENTS

0605115	7/1994	European Pat. Off. .
0653884	5/1995	European Pat. Off. .
0676898	10/1995	European Pat. Off. .
0746158	12/1996	European Pat. Off. .

OTHER PUBLICATIONS

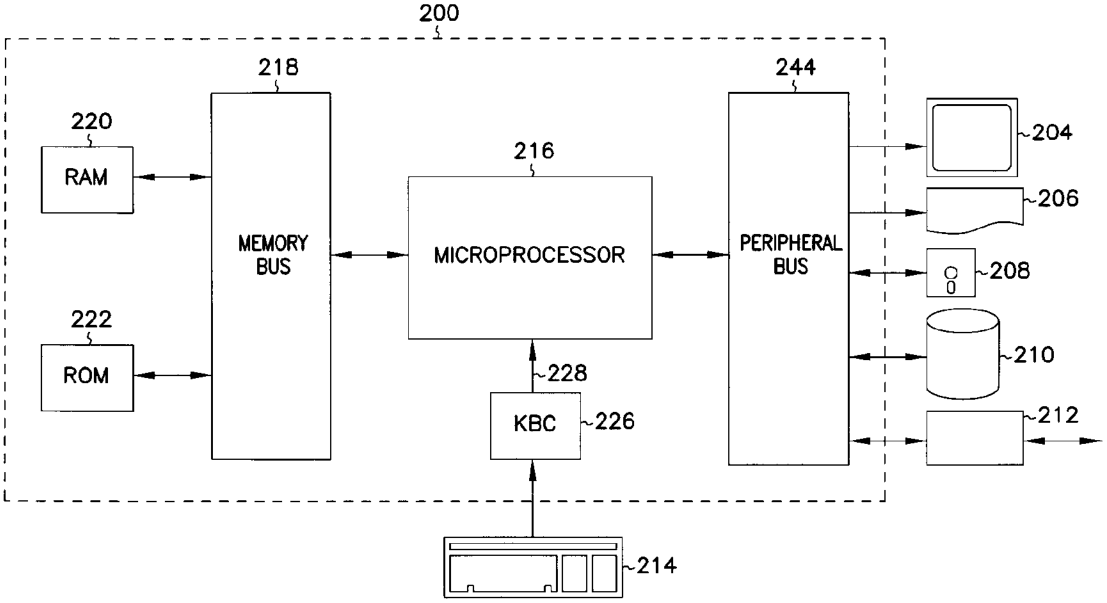
Chen, H.J., et al., "A Scalable Video–on–Demand Service for the Provision of VCR–Like Functions", IEEE Proceedings of the International Conference on Multimedia Computing and Systems, Washington, DC, 65–72, (May 15–18, 1995).

Primary Examiner—Victor R. Kostak
Attorney, Agent, or Firm—Schwegman, Lundberg, Woessner & Kluth, P.A.

[57] **ABSTRACT**

A method for employing a Hypertext Transfer Protocol (HTTP protocol) for transmitting streamed digital media data from a server. The server is configured for coupling to a client computer via a computer network. The method includes receiving at the server from the client an HTTP POST request. The POST request requests a first portion of the digital media data and includes a request header and a request entity-body. The request entity body includes a media command for causing the first portion of the digital media data to be sent from the server to the client. The method further includes sending an HTTP response to the client from the server. The HTTP response includes a response header and a response entity body. The response entity body includes at least a portion of the first portion of the digital media data.

35 Claims, 11 Drawing Sheets



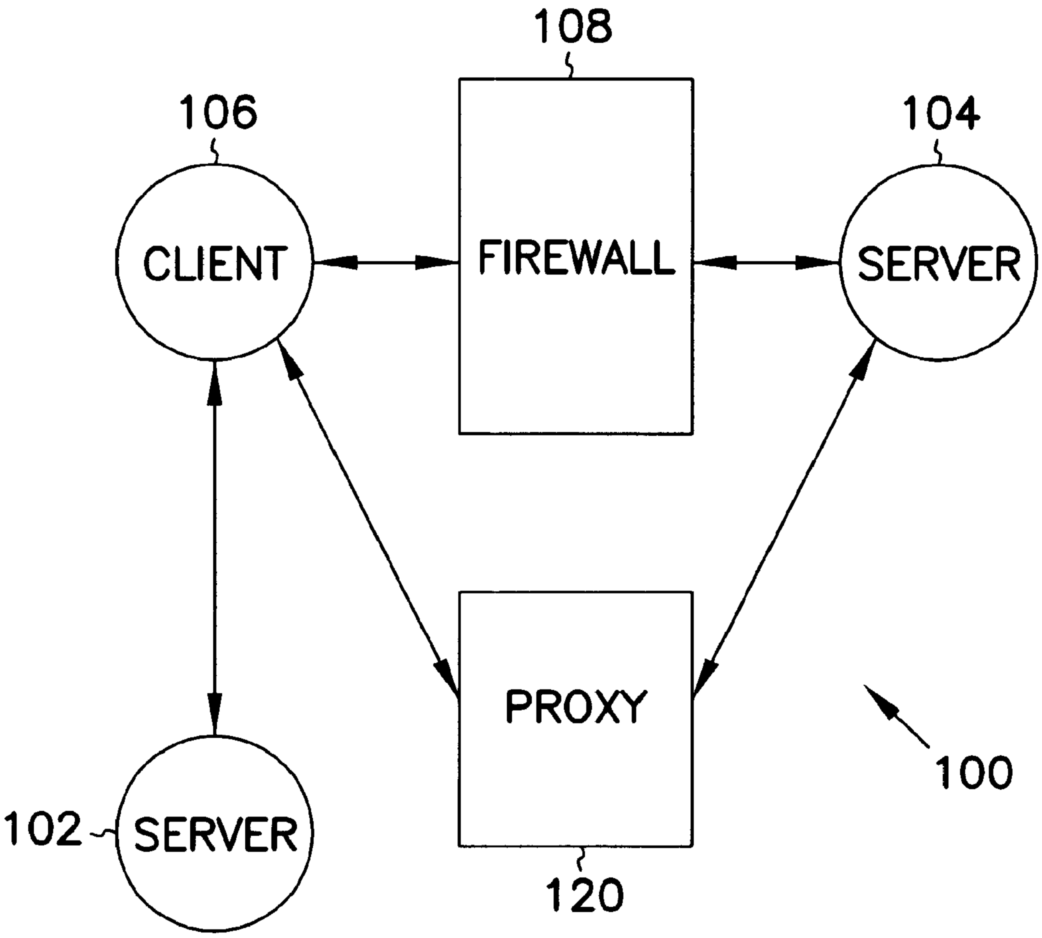


FIG. 1

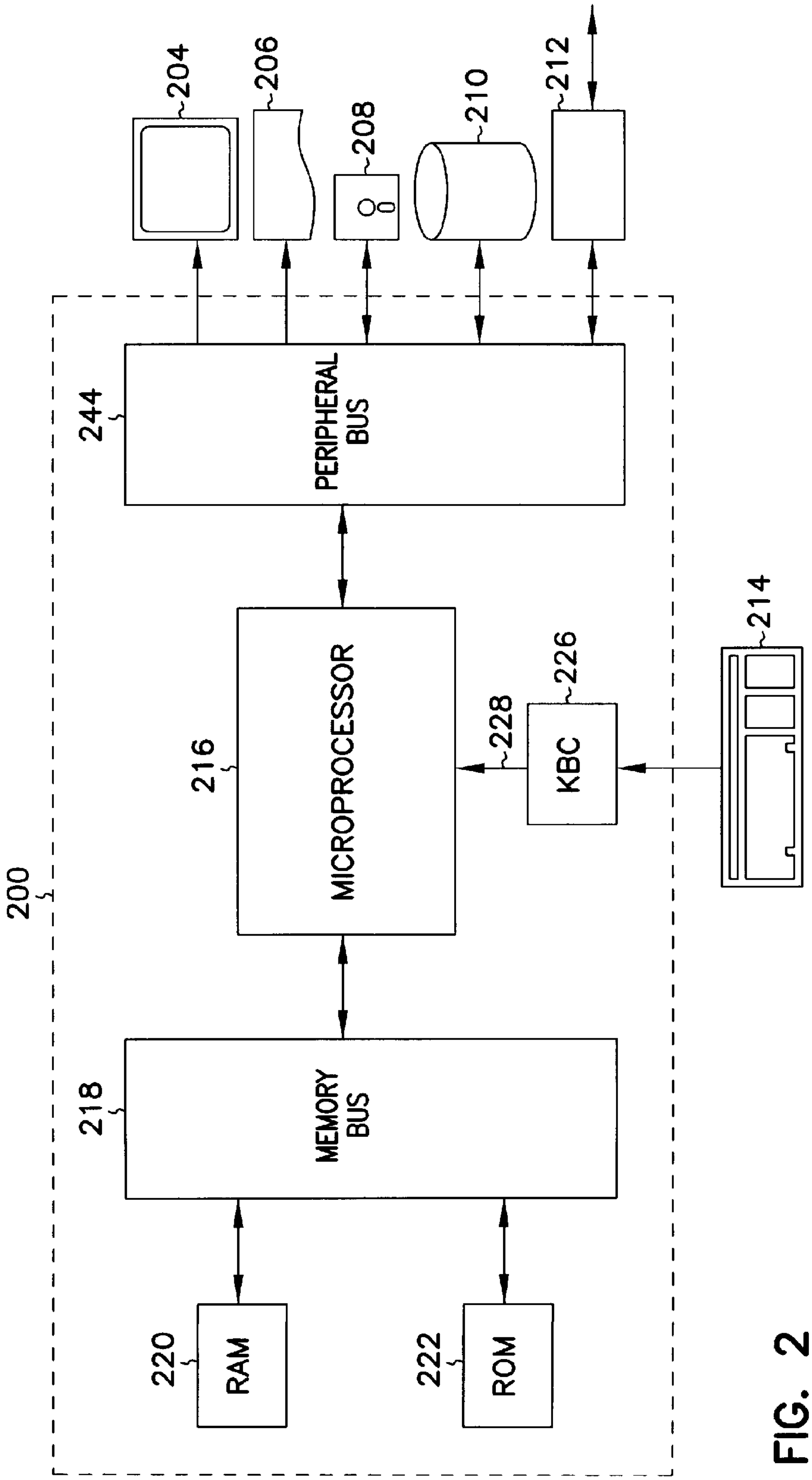

204
206
208
210
212
244
PERIPHERAL BUS
216
MICROPROCESSOR
228
226
KBC
214
200
218
MEMORY BUS
220
RAM
222
ROM

FIG. 2

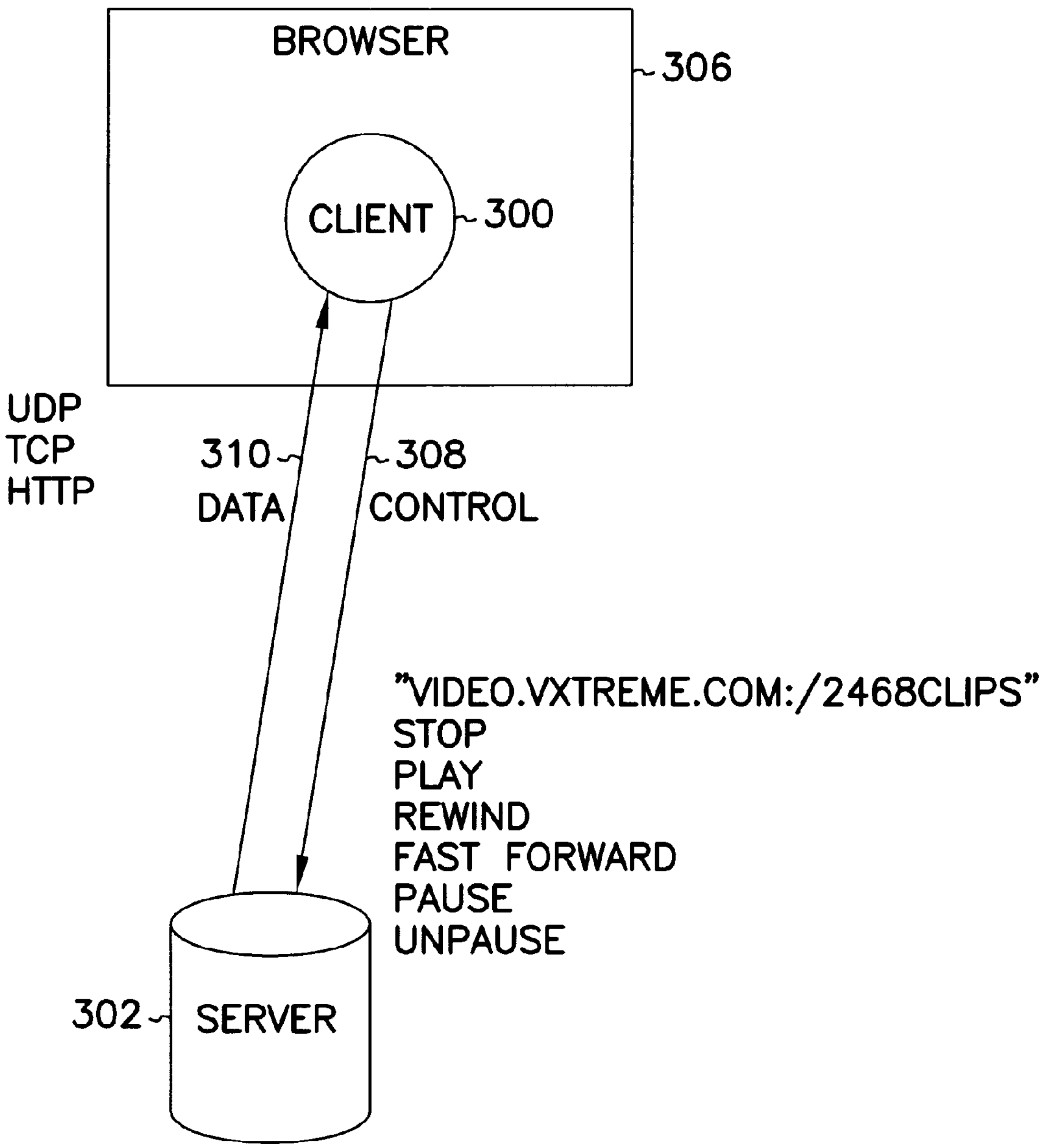

BROWSER
306
CLIENT
300
UDP
TCP
HTTP
310
DATA
308
CONTROL
"VIDEO.VXTREME.COM:/2468CLIPS"
STOP
PLAY
REWIND
FAST FORWARD
PAUSE
UNPAUSE
302
SERVER

FIG. 3

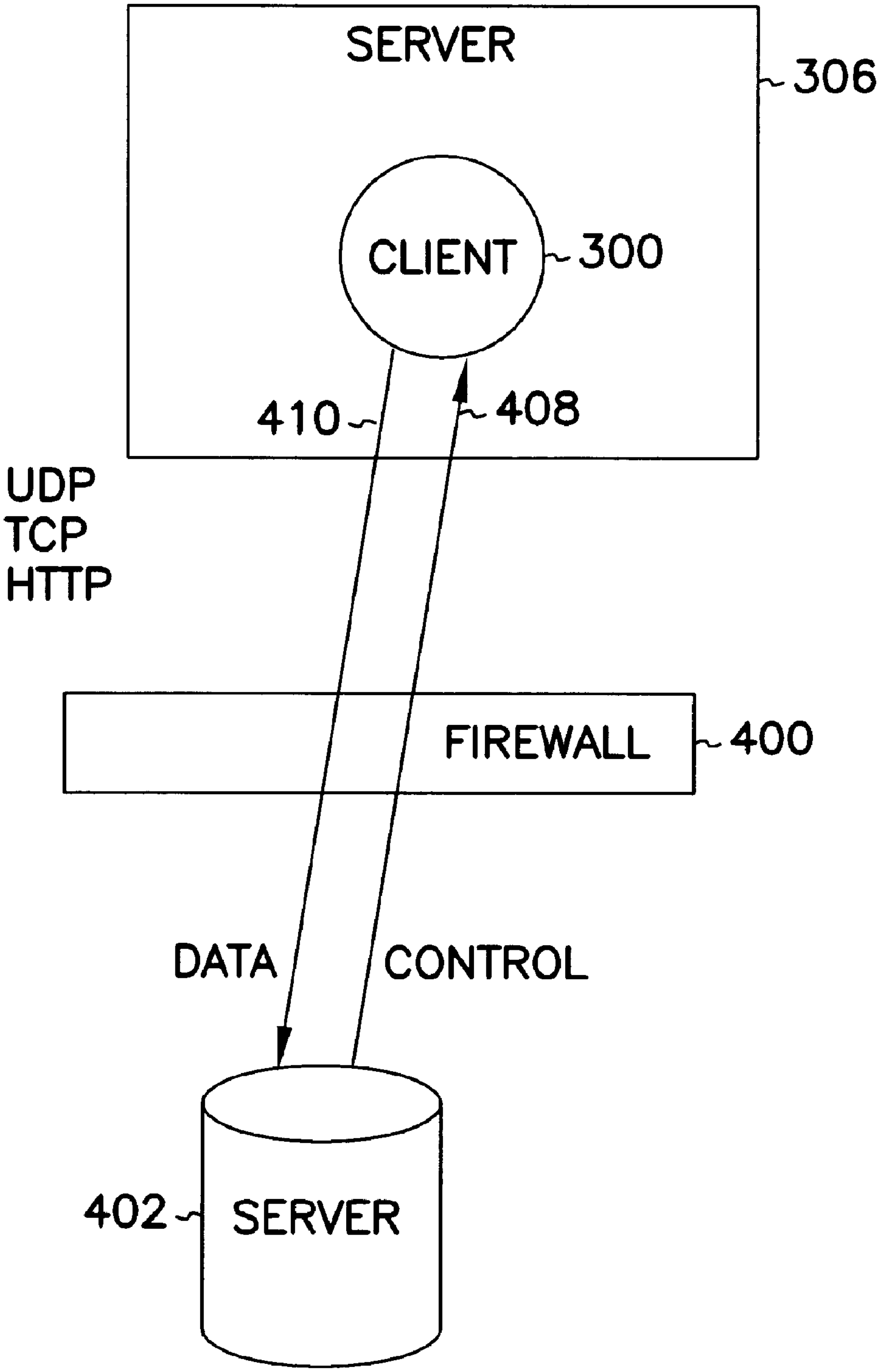

SERVER
306
CLIENT
300
410
408
UDP
TCP
HTTP
FIREWALL
400
DATA
CONTROL
402
SERVER

FIG. 4

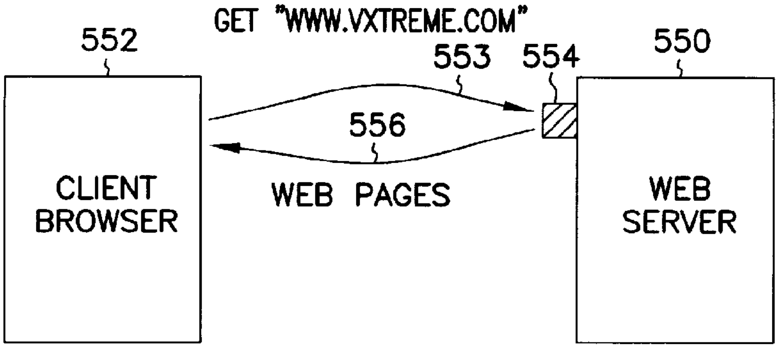


FIG. 5A (PRIOR ART)

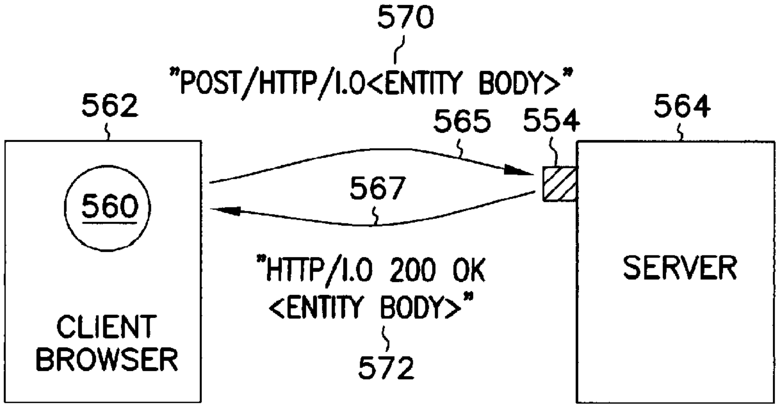


FIG. 5B

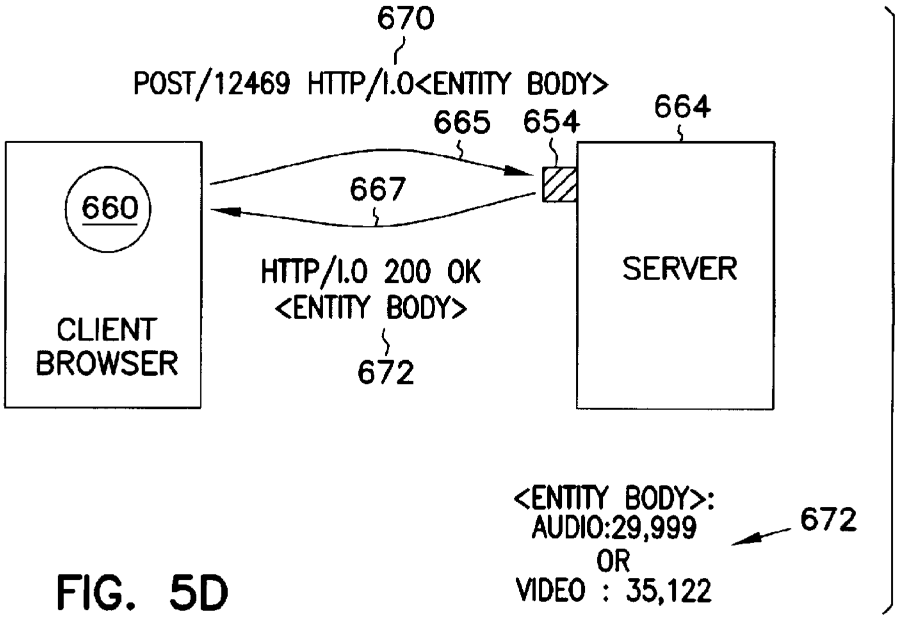


FIG. 5D

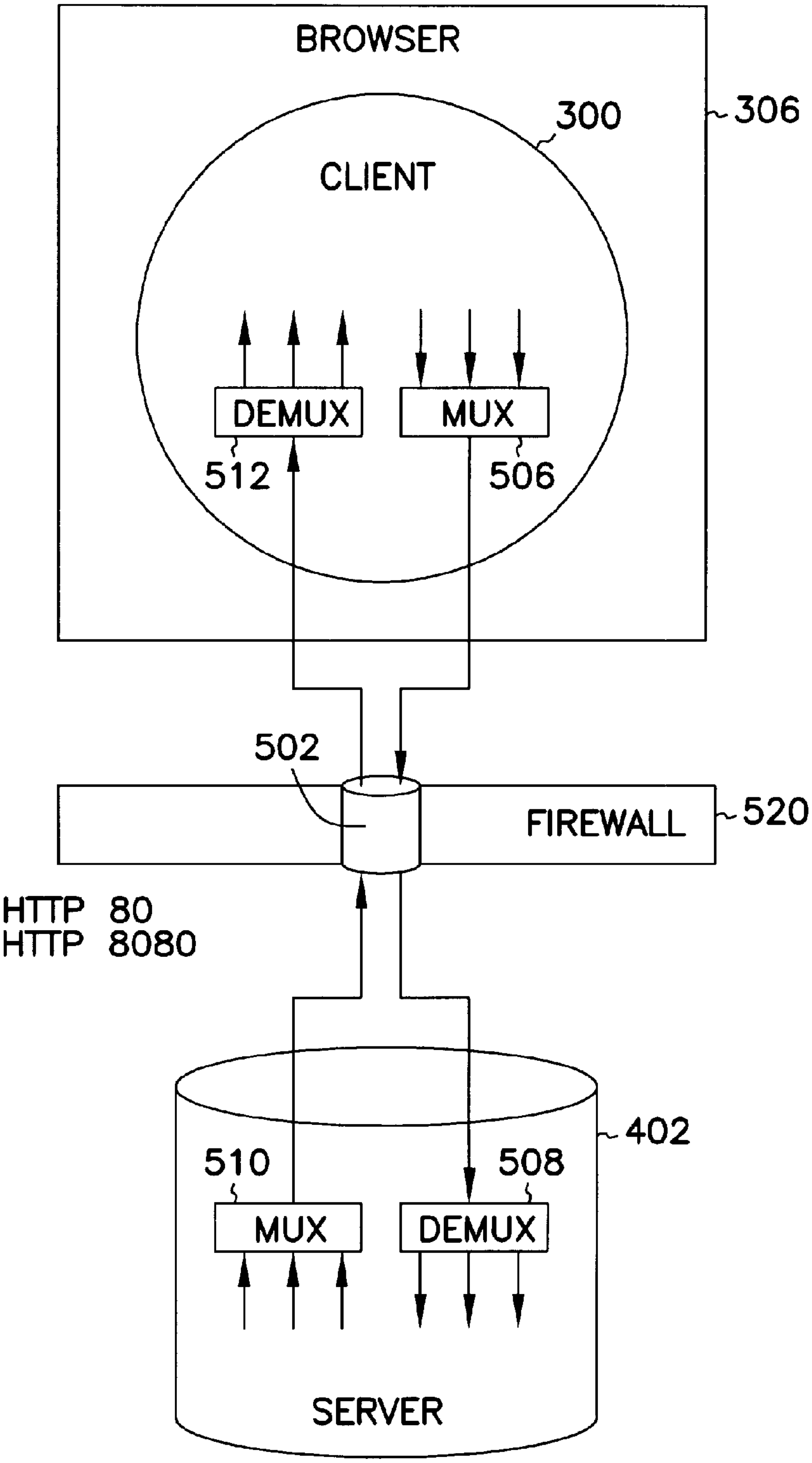

BROWSER
300
306
CLIENT
DEMUX
MUX
512
506
502
FIREWALL
520
HTTP 80
HTTP 8080
402
510
508
MUX
DEMUX
SERVER

FIG. 5C

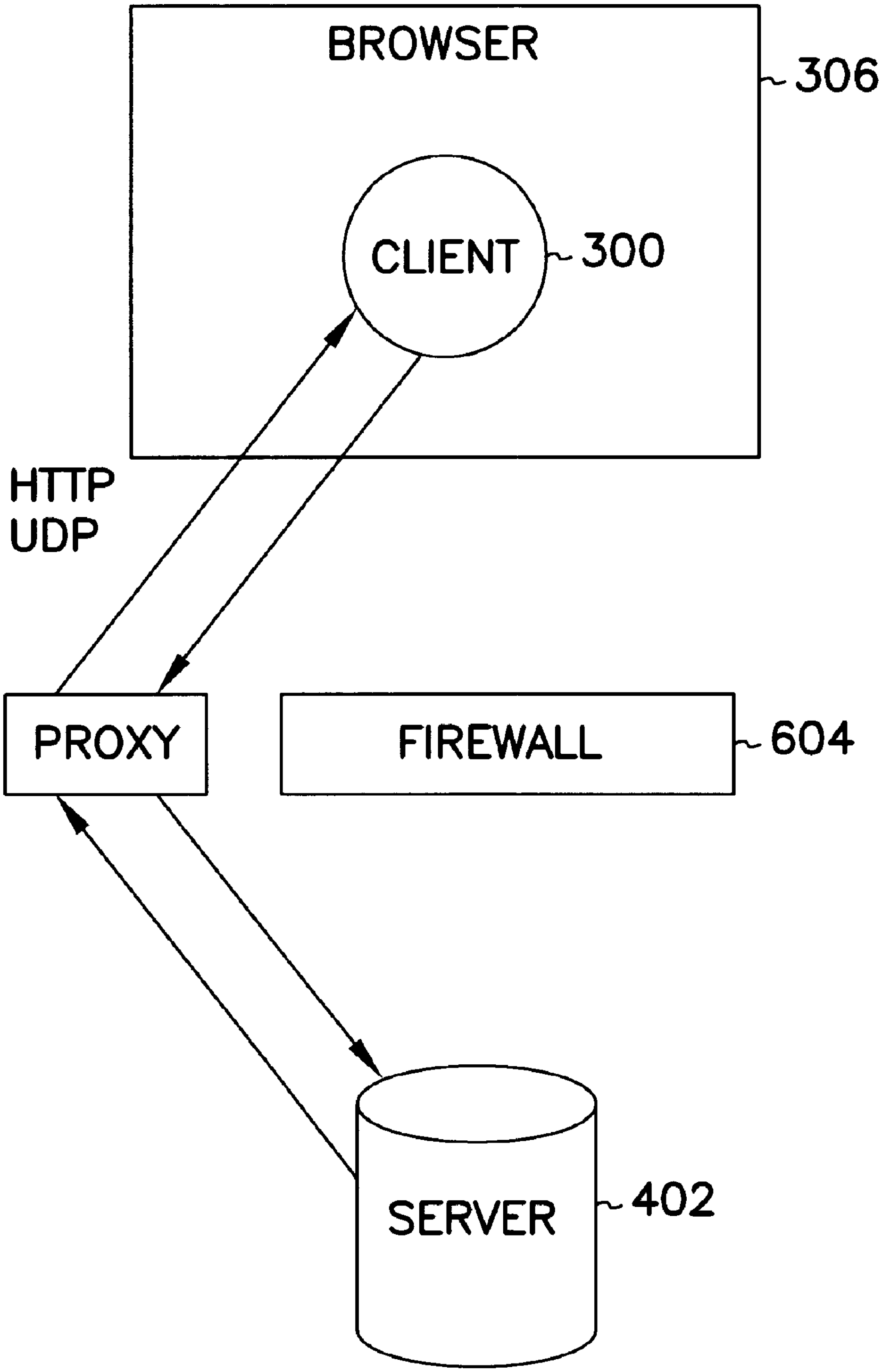

BROWSER
306
CLIENT
300
HTTP
UDP
PROXY
FIREWALL
604
SERVER
402

FIG. 6

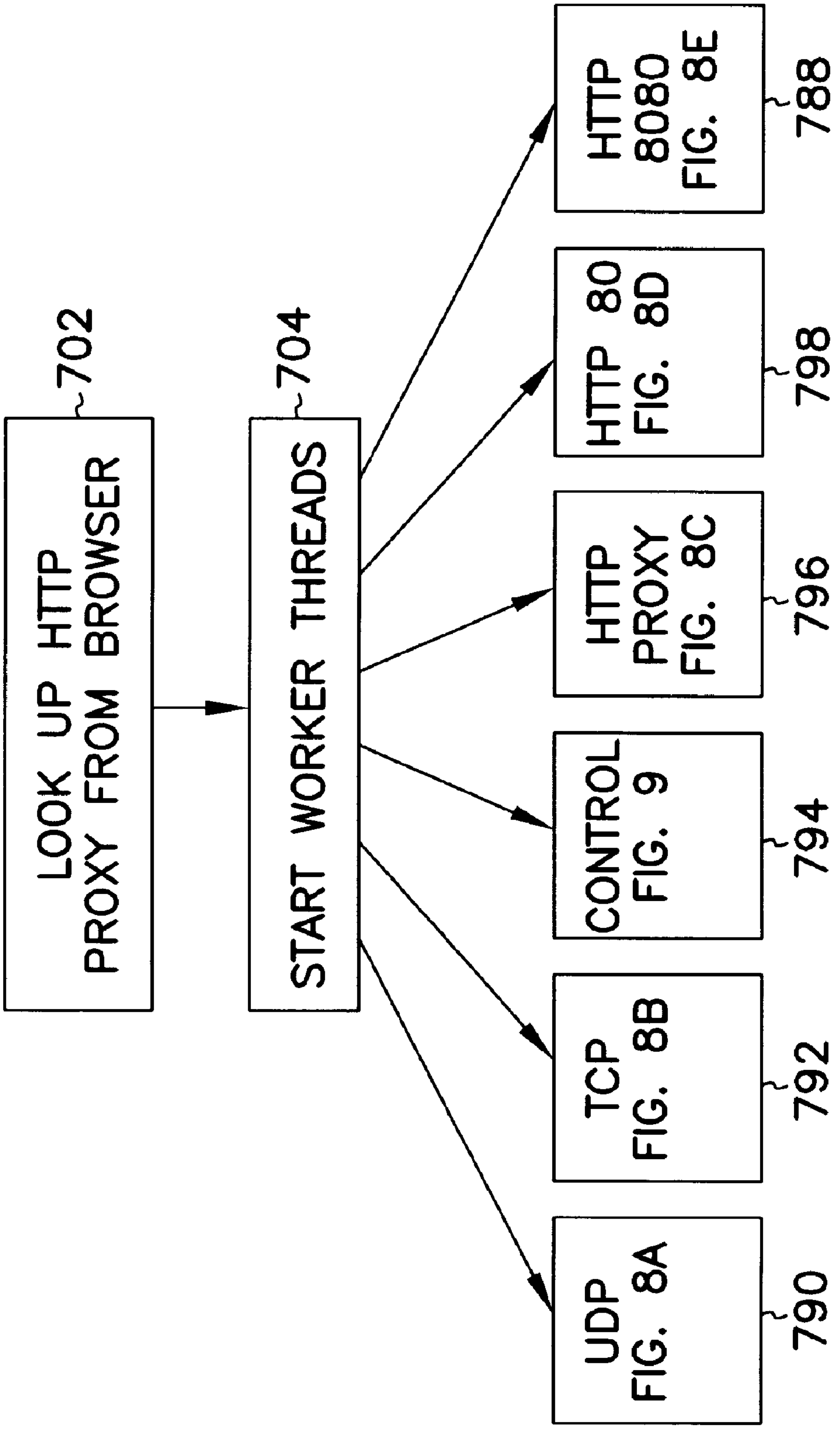

LOOK UP HTTP PROXY FROM BROWSER
702
START WORKER THREADS
704
UDP FIG. 8A
790
TCP FIG. 8B
792
CONTROL FIG. 9
794
HTTP PROXY FIG. 8C
796
HTTP 80 FIG. 8D
798
HTTP 8080 FIG. 8E
788

FIG. 7

FIG. 8A

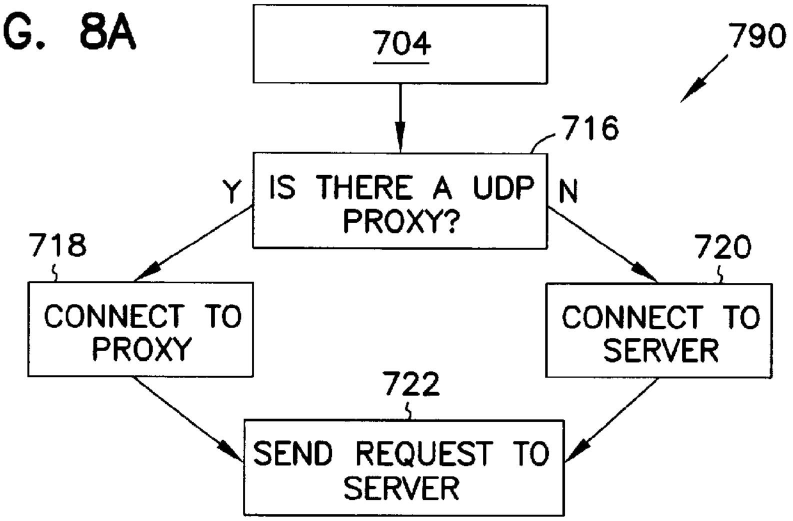


FIG. 8B

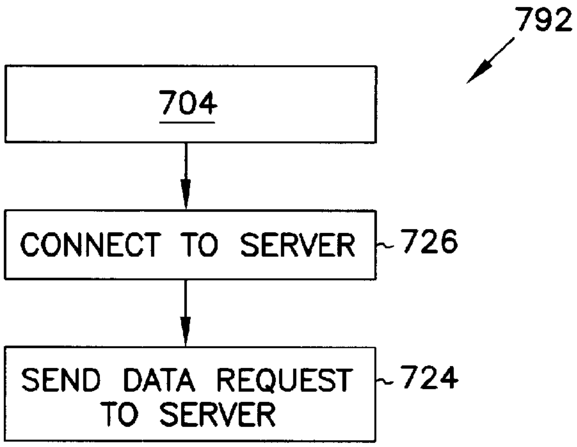


FIG. 8C

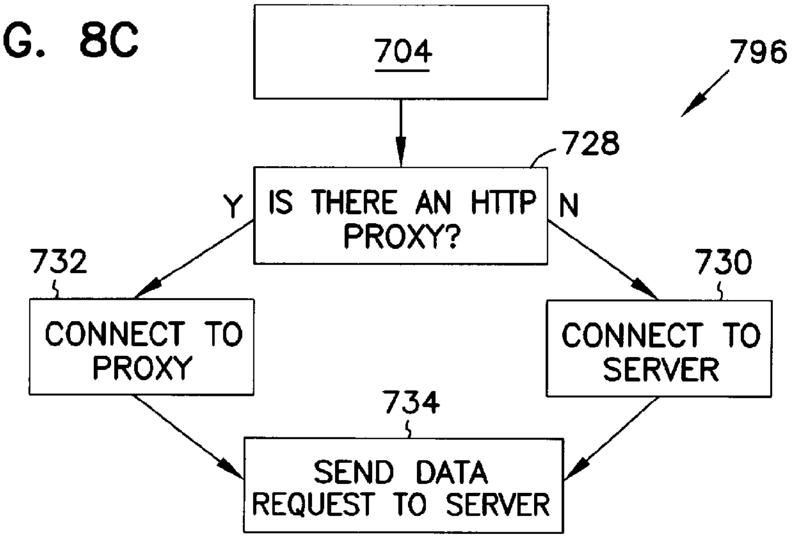


FIG. 8D

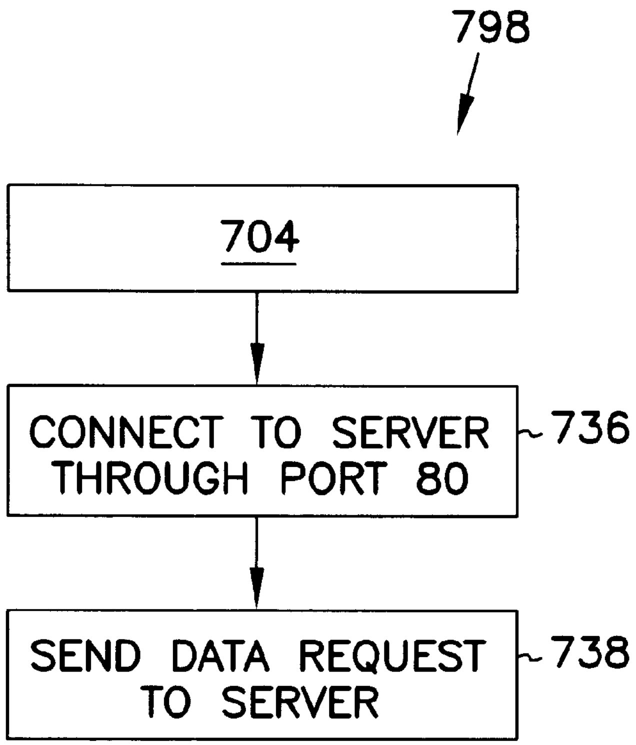
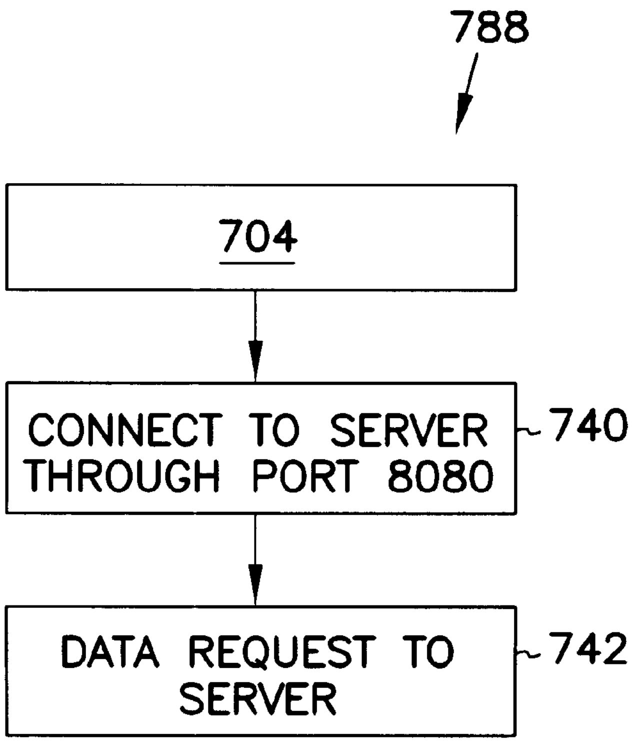

798
704
CONNECT TO SERVER THROUGH PORT 80
736
SEND DATA REQUEST TO SERVER
738

FIG. 8E


788
704
CONNECT TO SERVER THROUGH PORT 8080
740
DATA REQUEST TO SERVER
742

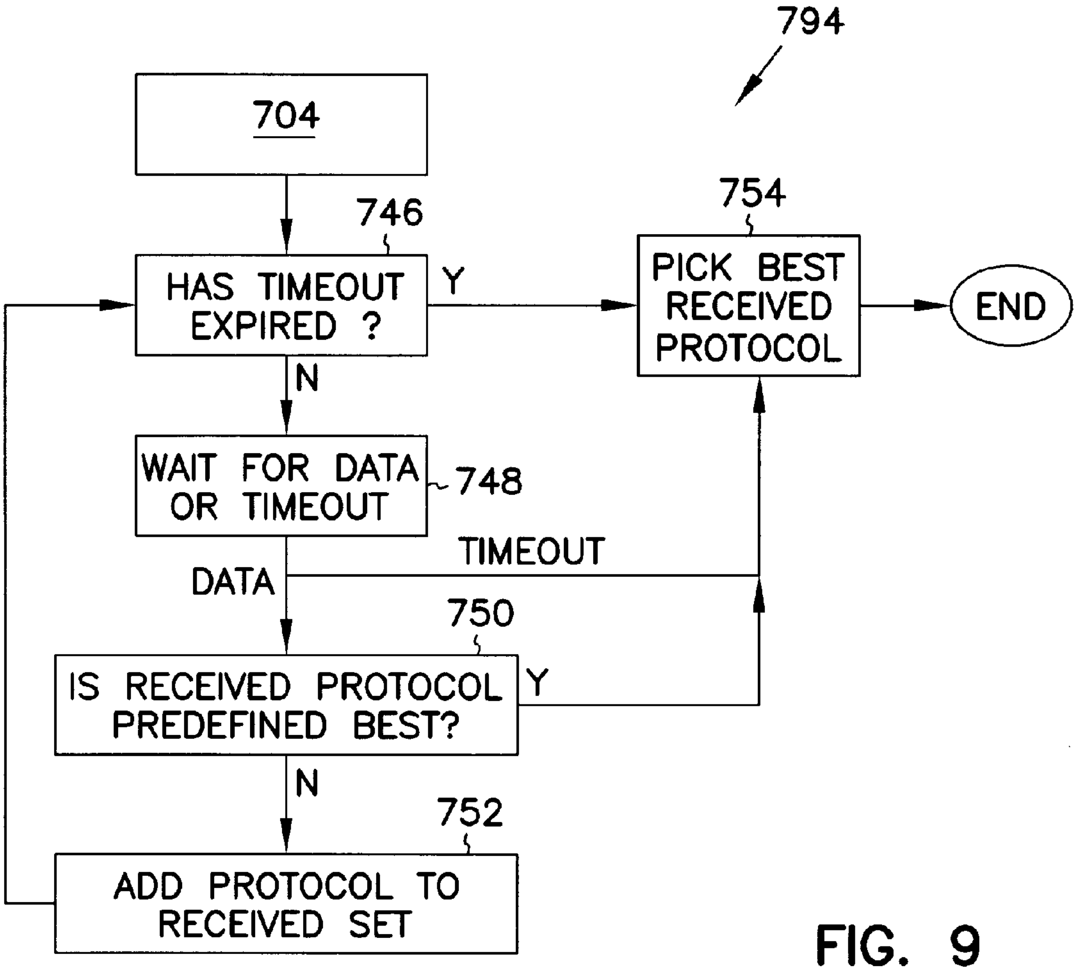

794
704
746
HAS TIMEOUT EXPIRED ?
Y
N
754
PICK BEST RECEIVED PROTOCOL
END
WAIT FOR DATA OR TIMEOUT
748
TIMEOUT
DATA
750
IS RECEIVED PROTOCOL PREDEFINED BEST?
Y
N
752
ADD PROTOCOL TO RECEIVED SET

FIG. 9

METHOD AND APPARATUS FOR COMMUNICATION MEDIA COMMANDS AND MEDIA DATA USING THE HTTP PROTOCOL

This application is related to co-pending U.S. application Ser. No. 08/818,805, filed on Mar. 14, 1997, entitled "Method and Apparatus for Implementing Motion Detection in Video Compression", U.S. application Ser. No. 08/819,507, filed on Mar. 14, 1997, entitled "Digital Video Signal Encoder and Encoding Method, U.S. application Ser. No. 08/818,804, filed on Mar. 14, 1997, entitled "Production of a Video Stream with Synchronized Annotations over a Computer Network, U.S. application Ser. No. 08/819,586, filed on Mar. 14, 1997, entitled "Method and Apparatus for Implementing Control Functions in a Streamed Video Display System, U.S. application Ser. No. 08/818,769, filed on Mar. 14, 1997, entitled "Method and Apparatus for Automatically Detecting Protocols in a Computer Network", U.S. application Ser. No. 08/818,127, filed on Mar. 14, 1997, entitled "Dynamic Bandwidth Selection for Efficient Transmission of Multimedia Streams in a Computer Network", U.S. application Ser. No. 08/819,585, filed on Mar. 14, 1997, entitled "Streaming and Display of a Video Stream with Synchronized Annotations over a Computer Network", U.S. application Ser. No. 08/818,664, filed on Mar. 14, 1997, entitled "Selective Retransmission for Efficient and Reliable Streaming of Multimedia Packets in a Computer Network", U.S. application Ser. No. 08/819,579, filed on Mar. 14, 1997, entitled "Method and Apparatus for Table-Based Compression with Embedded Coding", U.S. application Ser. No. 08/819,587, filed on Mar. 14, 1997, entitled "Method and Apparatus for Implementing Motion Estimation in Video Compression", U.S. application Ser. No. 08/818,826, filed on Mar. 14, 1997, entitled "Digital Video Signal Encoder and Encoding Method", all filed concurrently herewith, U.S. application Ser. No. 08/822,156, filed on Mar. 17, 1997, entitled "Method and Apparatus for Communication Media Commands and Data Using the HTTP Protocol", provisional U.S. application Ser. No. 60/036,662, filed on Jan. 30, 1997, entitled "Methods and Apparatus for Autodetecting Protocols in a Computer Network" U.S. application Ser. No. 08/625,650, filed on Mar. 29, 1996, entitled "Table-Based Low-Level Image Classification System", U.S. application Ser. No. 08/714,447, filed on Sep. 16, 1996, entitled "Multimedia Compression System with Additive Temporal Layers", and is a continuation-in-part of U.S. application Ser. No. 08/623,299, filed on Mar. 28, 1996, entitled "Table-Based Compression with Embedded Coding", which are all incorporated by reference in their entirety for all purposes.

BACKGROUND OF THE INVENTION

The present invention relates to data communication in a computer network. More particularly, the present invention relates to improved methods and apparatus for permitting a client computer in a client-server architecture computer network to exchange media commands and media data with the server using the HTTP (hypertext transfer protocol) protocol.

Client-server architectures are well known to those skilled in the computer art. For example, in a typical computer network, one or more client computers may be coupled to any number of server computers. Client computers typically refer to terminals or personal computers through which end users interact with the network. Server computers typically represent nodes in the computer network where data, application programs, and the like, reside. Server computers may also represent nodes in the network for forwarding data, programs, and the likes from other servers to the requesting client computers.

To facilitate discussion, FIG. **1** illustrates a computer network **100**, representing for example a subset of an international computer network popularly known as the Internet. As is well known, the Internet represents a well-known international computer network that links, among others, various military, governmental, educational, nonprofit, industrial and financial institutions, commercial enterprises, and individuals. There are shown in FIG. **1** a server **102**, a server **104**, and a client computer **106**. Server computer **104** is separated from client computer **106** by a firewall **108**, which may be implemented in either software or hardware, and may reside on a computer and/or circuit between client computer **106** and server computer **104**.

Firewall **108** may be specified, as is well known to those skilled in the art, to prevent certain types of data and/or protocols from traversing through it. The specific data and/or protocols prohibited or permitted to traverse firewall **108** depend on the firewall parameters, which are typically set by a system administrator responsible for the maintenance and security of client computer **106** and/or other computers connected to it, e.g., other computers in a local area network. By way of example, firewall **108** may be set up to prevent TCP, UDP, or HTTP (Transmission Control Protocol, User Datagram Protocol, and Hypertext Transfer Protocol, respectively) data and/or other protocols from being transmitted between client computer **106** and server **104**. The firewalls could be configured to allow specific TCP or UDP sessions, for example outgoing TCP connection to certain ports, UDP sessions to certain ports, and the like.

Without a firewall, any type of data and/or protocol may be communicated between a client computer and a server computer if appropriate software and/or hardware are employed. For example, server **102** resides on the same side of firewall **108** as client computer **106**, i.e., firewall **108** is not disposed in between the communication path between server **102** and client computer **106**. Accordingly, few, if any, of the protocols that client computer **106** may employ to communicate with server **102** may be blocked.

As is well known to those skilled in the art, some computer networks may be provided with proxies, i.e., software codes or hardware circuitries that facilitate the indirect communication between a client computer and a server around a firewall. With reference to FIG. **1**, for example, client computer **106** may communicate with server **104** through proxy **120**. Through proxy **120**, HTTP data, which may otherwise be blocked by firewall **108** for the purpose of this example, may be transmitted between client computer **106** and server computer **104**.

In the prior art, the HTTP protocol is typically employed to transmit web pages between the client computer and the server computer. As is well known to those skilled, the HTTP protocol, as specified by for example the Internet Request For Comments RFC 1945 (T. Berners-Lee et al.), typically defines only three types of requests to be sent from the client computer to the server, namely GET, POST, and HEAD. The POST command, for instance, is specified in RFC 1945 to be composed of a Request-Line, one or more Headers and Entity-Body.

In some cases, it may be desirable, however, to employ the HTTP protocol to communicate other types of commands and receive other types of data between the client computer and the server computer. By way of example, in applications such as real-time or live video streaming, the

HTTP protocol may represent, on some networks, the most advantageous protocol available for use in transmitting and receiving data. This is because, for example, there may exists firewalls or other network limitations that inhibit the use of other protocols for transmitting media control commands and for receiving media data. Media control commands may represent, for example, commands to fast forward on the play stream, to seek backward on the play stream, to begin playing at a certain frame, to stop, to pause, and the like. Media data may represent, for example, realtime or live video, audio, or annotation data. In these cases, the ability to use the HTTP protocol to transmit media commands and to receive media data may indeed be valuable.

In view of the foregoing, there are desired improved techniques for permitting a client computer in a client-server architecture computer network to exchange media commands and media a data with the server computer using the HTTP (hypertext transfer protocol) protocol.

SUMMARY OF THE INVENTION

The invention relates, in one embodiment, to a method for employing a Hypertext Transfer Protocol (HTTP protocol) for transmitting streamed digital media data from a server. The server is configured for coupling to a client computer via a computer network. The method includes receiving at the server from the client an HTTP POST request. The POST request requests a first portion of the digital media data and includes a request header and a request entity-body. The request entity body includes a media command for causing the first portion of the digital media data to be sent from the server to the client. The method further includes sending an HTTP response to the client from the server. The HTTP response includes a response header and a response entity body. The response entity body includes at least a portion of the first portion of the digital media data.

In another embodiment, the invention relates to a computer readable medium containing computer readable instructions for transmitting streamed media data employing a Hypertext Transfer Protocol (HTTP protocol) from a server. The server is configured for coupling to a client computer via a computer network. The method includes receiving at the server from the client an HTTP POST request. The POST request requests a first portion of the digital media data and includes a request header and a request entity-body. The request entity body includes a media command for causing the first portion of the digital media data to be sent from the server to the client. The method further includes sending an HTTP response to the client from the server. The HTTP response includes a response header and a response entity body. The response entity body includes at least a portion of the first portion of the digital media data.

These and other features of the present invention will be described in more detail below in the detailed description of the invention and in conjunction with the following figures.

BRIEF DESCRIPTION OF THE DRAWINGS

To facilitate discussion, FIG. **1** illustrates a computer network, representing for example a portion of an international computer network popularly known as the Internet.

FIG. **2** is a block diagram of an exemplar computer system for carrying out the autodetect technique according to one embodiment of the invention.

FIG. **3** illustrates, in accordance with one embodiment, the control and data connections between a client application and a server computer when no firewall is provided in the network.

FIG. **4** illustrates another network arrangement wherein control and data connections are established through a firewall.

FIGS. **5**A–B illustrates another network arrangement wherein media control commands and media data may be communicated between a client computer and a server computer using the HTTP protocol.

FIGS. **5**C–D illustrate another network arrangement wherein multiple HTTP control and data connections are multiplexed through a single HTTP port.

FIG. **6** illustrates another network arrangement wherein control and data connections are transmitted between the client application and the server computer via a proxy.

FIG. **7** depicts, in accordance with one embodiment of the present invention, a simplified flowchart illustrating the steps of the inventive autodetect technique.

FIG. **8**A depicts, in accordance with one aspect of the present invention, the steps involved in executing the UDP protocol thread of FIG. **7**.

FIG. **8**B depicts, in accordance with one aspect of the present invention, the steps involved in executing the TCP protocol thread of FIG. **7**.

FIG. **8**C depicts, in accordance with one aspect of the present invention, the steps involved in executing the HTTP protocol thread of FIG. **7**.

FIG. **8**D depicts, in accordance with one aspect of the present invention, the steps involved in executing the HTTP **80** protocol thread of FIG. **7**.

FIG. **8**E depicts, in accordance with one aspect of the present invention, the steps involved in executing the HTTP **8080** protocol thread of FIG. **7**.

FIG. **9** illustrates, in accordance with one embodiment of the present invention, the steps involved in executing the control thread of FIG. **7**.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

The present invention will now be described in detail with reference to a few preferred embodiments thereof as illustrated in the accompanying drawings. In the following description, numerous specific details are set forth in order to provide a thorough understanding of the present invention. It will be apparent, however, to one skilled in the art, that the present invention may be practiced without some or all of these specific details. In other instances, well known process steps have not been described in detail in order to not unnecessarily obscure the present invention.

In accordance with one aspect of the present invention, the client computer in a heterogeneous client-server computer network (e.g., client computer **106** in FIG. **1**) is provided with an autodetect mechanism. When executed, the autodetect mechanism advantageously permits client computer **106** to select, in an efficient and automatic manner, the most advantageous protocol for communication between the client computer and its server. Once the most advantageous protocol is selected, parameters pertaining to the selected protocol are saved to enable the client computer, in future sessions, to employ the same selected protocol for communication.

In accordance with one particular advantageous embodiment, the inventive autodetect mechanism simultaneously employs multiple threads, through multiple connections, to initiate communication with the server computer, e.g., server **104**. Each thread preferably employs a different protocol and requests the server computer to respond to the client computer using the protocol associated with that thread. For example, client computer **106** may, employing the autodetect mechanism, initiate five different threads, using respectively the TCP, UDP, one of HTTP and HTTP proxy, HTTP through port (multiplex) **80**, and HTTP through port (multiplex) **8080** protocols to request server **104** to respond.

Upon receiving a request, server **104** responds with data using the same protocol as that associated with the thread on which the request arrives. If one or more protocols is blocked and fails to reach server **104** (e.g., by a firewall), no response employing the blocked protocol would of course be transmitted from server **104** to client computer **106**. Further, some of the protocols transmitted from server **104** to client computer **106** may be blocked as well. Accordingly, client computer may receive only a subset of the responses sent from server **104**.

In one embodiment, client computer **106** monitors the set of received responses. If the predefined "best" protocol is received, that protocol is then selected for communication by client computer **106**. The predefined "best" protocol may be defined in advance by the user and/or the application program. If the predefined "best" protocol is, however, blocked (as the request is transmitted from the client computer or as the response is transmitted from the server, for example) the most advantageous protocol may simply be selected from the set of protocols received back by the client computer. In one embodiment, the selection may be made among the set of protocols received back by the client computer within a predefined time period after the requests are sent out in parallel.

The selection of the most advantageous protocol for communication among the protocols received by client computer **106** may be performed in accordance with some predefined priority. For example, in the real-time data rendering application, the UDP protocol may be preferred over TCP protocol, which may be in turned preferred over the HTTP protocol. This is because UDP protocol typically can handle a greater data transmission rate and may allow client computer **106** to exercise a greater degree of control over the transmission of data packets.

HTTP data, while popular nowadays for use in transmitting web pages, typically involves a higher number of overhead bits, making it less efficient relative to the UDP protocol for transmitting real-time data. As is known, the HTTP protocol is typically built on top of TCP. The underlaying TCP protocol typically handles the transmission and retransmission requests of individual data packets automatically. Accordingly, the HTTP protocol tends to reduce the degree of control client computer **106** has over the transmission of the data packets between server **104** and client computer **106**. Of course other priority schemes may exist for different applications, or even for different real-time data rendering applications.

In one embodiment, as client computer **106** is installed and initiated for communication with server **104** for the first time, the autodetect mechanism is invoked to allow client computer **106** to send transmission requests in parallel (e.g., using different protocols over different connections) in the manner discussed earlier. After server **104** responds with data via multiple connections/protocols and the most advantageous protocol has been selected by client computer **106** for communication (in accordance with some predefined priority), the parameters associated with the selected protocol are then saved for future communication.

Once the most advantageous protocol is selected, the autodetect mechanism may be disabled, and future communication between client computer **106** and server **104** may proceed using the selected most advantageous protocol without further invocation of the autodetect mechanism. If the topology of computer network **100** changes and communication using the previously selected "most advantageous" protocol is no longer appropriate, the autodetect mechanism may be executed again to allow client computer **106** to ascertain a new "most advantageous" protocol for communication with server **104**. In one embodiment, the user of client computer **106** may, if desired, initiate the autodetect mechanism at anytime in order to enable client computer **106** to update the "most advantageous" protocol for communication with server **104** (e.g., when the user of client computer **106** has reasons to suspect that the previously selected "most advantageous" protocol is no longer the most optimal protocol for communication).

The inventive autodetect mechanism may be implemented either in software or hardware, e.g., via an IC chip. If implemented in software, it may be carried out by any number of computers capable of functioning as a client computer in a computer network. FIG. **2** is a block diagram of an exemplar computer system **200** for carrying out the autodetect technique according to one embodiment of the invention. Computer system **200**, or an analogous one, may be employed to implement either a client or a server of a computer network. The computer system **200** includes a digital computer **202**, a display screen (or monitor) **204**, a printer **206**, a floppy disk drive **208**, a hard disk drive **210**, a network interface **212**, and a keyboard **214**. The digital computer **202** includes a microprocessor **216**, a memory bus **218**, random access memory (RAM) **220**, read only memory (ROM) **222**, a peripheral bus **224**, and a keyboard controller **226**. The digital computer **200** can be a personal computer (such as an Apple computer, e.g., an Apple Macintosh, an IBM personal computer, or one of the compatibles thereof), a workstation computer (such as a Sun Microsystems or Hewlett-Packard workstation), or some other type of computer.

The microprocessor **216** is a general purpose digital processor which controls the operation of the computer system **200**. The microprocessor **216** can be a single-chip processor or can be implemented with multiple components. Using instructions retrieved from memory, the microprocessor **216** controls the reception and manipulation of input data and the output and display of data on output devices.

The memory bus **218** is used by the microprocessor **216** to access the RAM **220** and the ROM **222**. The RAM **220** is used by the microprocessor **216** as a general storage area and as scratch-pad memory, and can also be used to store input data and processed data. The ROM **222** can be used to store instructions or program code followed by the microprocessor **216** as well as other data.

The peripheral bus **224** is used to access the input, output, and storage devices used by the digital computer **202**. In the described embodiment, these devices include the display screen **204**, the printer device **206**, the floppy disk drive **208**, the hard disk drive **210**, and the network interface **212**, which is employed to connect computer **200** to the network. The keyboard controller **226** is used to receive input from keyboard **214** and send decoded symbols for each pressed key to microprocessor **216** over bus **228**.

The display screen **204** is an output device that displays images of data provided by the microprocessor **216** via the peripheral bus **224** or provided by other components in the computer system **200**. The printer device **206** when operating as a printer provides an image on a sheet of paper or a similar surface. Other output devices such as a plotter, typesetter, etc. can be used in place of, or in addition to, the printer device **206**.

The floppy disk drive **208** and the hard disk drive **210** can be used to store various types of data. The floppy disk drive **208** facilitates transporting such data to other computer systems, and hard disk drive **210** permits fast access to large amounts of stored data.

The microprocessor **216** together with an operating system operate to execute computer code and produce and use data. The computer code and data may reside on the RAM **220**, the ROM **222**, the hard disk drive **220**, or even on another computer on the network. The computer code and data could also reside on a removable program medium and loaded or installed onto the computer system **200** when needed. Removable program mediums include, for example, CD-ROM, PC-CARD, floppy disk and magnetic tape.

The network interface circuit **212** is used to send and receive data over a network connected to other computer systems. An interface card or similar device and appropriate software implemented by the microprocessor **216** can be used to connect the computer system **200** to an existing network and transfer data according to standard protocols.

The keyboard **214** is used by a user to input commands and other instructions to the computer system **200**. Other types of user input devices can also be used in conjunction with the present invention. For example, pointing devices such as a computer mouse, a track ball, a stylus, or a tablet can be used to manipulate a pointer on a screen of a general-purpose computer.

The invention can also be embodied as computer readable code on a computer readable medium. The computer readable medium is any data storage device that can store data which can be thereafter be read by a computer system. Examples of the computer readable medium include read-only memory, random-access memory, CD-ROMs, magnetic tape, optical data storage devices. The computer readable code can also be distributed over a network coupled computer systems so that the computer readable code is stored and executed in a distributed fashion.

FIGS. **3–6** below illustrate, to facilitate discussion, some possible arrangements for the transmission and receipt of data in a computer network. The arrangements differ depend on which protocol is employed and the configuration of the network itself. FIG. **3** illustrates, in accordance with one embodiment, the control and data connections between a client application **300** and server **302** when no firewall is provided in the network.

Client application **300** may represent, for example, the executable codes for executing a real-time data rendering program such as the Web Theater Client 2.0, available from VXtreme, Inc. of Sunnyvale, Calif. In the example of FIG. **3**, client application **300** includes the inventive autodetect mechanism and may represent a plug-in software module that may be installed onto a browser **306**. Browser **306** may represent, for example, the application program which the user of the client computer employs to navigate the network. By way of example, browser **306** may represent one of the popular Internet browser programs, such as Netscape™ by Netscape Communications Inc. of Mountain View, Calif. or Microsoft Explorer by Microsoft Corporation of Redmond, Wash.

When the autodetect mechanism of client application **300** is executed in browser **306** (e.g., during the set up of client application **300**), client application **300** sends a control request over control connection **308** to server **302**. Although multiple control requests are typically sent in parallel over multiple control connections using different protocols as discussed earlier, only one control request is depicted in FIG. **3** to facilitate ease of illustration.

The protocol employed to send the control request over control connection **308** may represent, for example, TCP, or HTTP. If UDP protocol is requested from the server, the request from the client may be sent via the control connection using for example the TCP protocol. Initially, each control request from client application **300** may include, for example, the server name that identifies server **302**, the port through which control connection may be established, and the name of the video stream requested by client application **300**. Server **302** then responds with data via data connection **310**.

In FIG. **3**, it is assumed that no proxies and/or firewalls exist. Accordingly, server **302** responds using the same protocol as that employed in the request. If the request employs TCP, however, server **302** may attempt to respond using either UDP or TCP data connections (depending on the specifics of the request). The response is sent to client application via data connection **310**. If the protocol received by the client application is subsequently selected to be the "most advantageous" protocol, subsequent communication between client application **300** and server **302** may take place via control connection **308** and data connection **310**. Subsequent control requests sent by client application **300** via control connection **308** may include, for example, stop, play, fast forward, rewind, pause, unpause, and the like. These control requests may be utilized by server **302** to control the delivery of the data stream from server **302** to client application **300** via data connection **310**.

It should be noted that although only one control connection and one data connection is shown in FIG. **3** to simplify illustration, multiple control and data connections utilizing the same protocol may exist during a data rendering session. Multiple control and data connections may be required to handle the multiple data streams (e.g., audio, video, annotation) that may be needed in a particular data rendering session. If desired, multiple clients applications **300** may be installed within browser **306**, e.g., to simultaneously render multiple video clips, each with its own sound and annotations.

FIG. **4** illustrates another network arrangement wherein control and data connections are established through a firewall. As mentioned earlier, a firewall may have policies that restrict or prohibit the traversal of certain types of data and/or protocols. In FIG. **4**, a firewall **400** is disposed between client application **300** and server **402**. Upon execution, client application **300** sends control request using a given protocol via firewall **400** to server **402**. Server **402** then responds with data via data connection **410**, again via firewall **400**.

If the data and/or protocol can be received by the client computer through firewall **400**, client application **300** may then receive data from server **402** (through data connection **408**) in the same protocol used in the request. As before, if the request employs the TCP protocol, the server may respond with data connections for either TCP or UDP protocol (depending on the specifics of the request). Protocols that may traverse a firewall may include one or more of the following: UDP, TCP, and HTTP.

In accordance with one aspect of the present invention, the HTTP protocol may be employed to send/receive media data (video, audio, annotation, or the like) between the client and the server. FIG. **5A** is a prior art drawing illustrating how a client browser may communicate with a web server using a port designated for communication. In FIG. 5, there is shown a web server **550**, representing the software module for serving web pages to a browser application **552**. Web server **550** may be any of the commercially available web servers that are available from, for example, Netscape Communications Inc. of Mountain View, Calif. or Microsoft Corporation of Redmond, Wash. Browser application **552** represents for example the Netscape browser from the aforementioned Netscape Communications, Inc., or similarly suitable browser applications.

Through browser application **552**, the user may, for example, obtain web pages pertaining to a particular entity by sending an HTTP request (e.g., GET) containing the URL (uniform resource locator) that identifies the web page file. The request sent via control connection **553** may arrive at web server **550** through the HTTP port **554**. HTTP port **554** may represent any port through which HTTP communication is enabled. HTTP port **554** may also represent the default port for communicating web pages with client browsers. The HTTP default port may represent, for example, either port **80** or port **8080** on web server **550**. As is known, one or both of these ports on web server **550** may be available for web page communication even if there are firewalls disposed between the web server **550** and client browser application **552**, which otherwise block all HTTP traffic in other ports. Using the furnished URL, web server **550** may then obtain the desired web page(s) for sending to client browser application **552** via data connection **556**.

The invention, in one embodiment, involves employing the HTTP protocol to communicate media commands from a browser application or browser plug-in to the server. Media commands are, for example, PLAY, STOP, REWIND, FAST FORWARD, and PAUSE. The server computer may represent, for example, a web server. The server computer may also represent a video server for streaming video to the client computer. Through the use of the HTTP protocol the client computer may successfully send media control requests and receive media data through any HTTP port. If the default HTTP port, e.g., port **80** or **8080**, is specified, the client may successfully send media control requests and receive media data even if there exists a firewall or an HTTP Proxy disposed in between the server computer and the client computer, which otherwise blocks all other traffic that does not use the HTTP protocol. For example, these firewalls or HTTP Proxies do not allow regular TCP or UDP packets to go through.

As is well known to those skilled, the HTTP protocol, as specified by for example the Internet Request For Comments RFC 1945 (T. Berners-Lee et al.), typically defines only three types of requests to be sent from the client computer to the server, namely GET, POST, and HEAD. The POST command, for instance, is specified in RFC 1945 to be composed of a Request-Line, one or more Headers and Entity-Body. To send media commands like PLAY, REWIND, etc., the invention in one embodiment sends the media command as part of the Entity-Body of the HTTP POST command. The media command can be in any format or protocol, and can be, for instance, in the same format as that employed when firewalls are not a concern and plain TCP protocol can be used. This format can be, for example, RTSP (Real Time Streaming Protocol).

When a server gets an HTTP request, it answers the client with an HTTP Response. Responses are typically composed of a Status-Line, one or more headers, and an Entity-Body. In one embodiment of this invention, the response to the media commands is sent as the Entity-Body of the response to the original HTTP request that carried the media command.

FIG. 5B illustrates this use of HTTP for sending arbitrary media commands. In FIG. 5B, the plug-in application **560** within client browser application **562** may attempt to receive media data (e.g., video, audio, annotation, or the like) by first sending an HTTP request to server **564** via control connection **565**. For example, a REWIND command could be sent from the client **560** to the server **564** as an HTTP packet **570** of the form: "POST/HTTP/1.0<Entity-Body containing rewind command in any suitable media protocol>". The server can answer to this request with an HTTP response **572** of the form: "HTTP/1.0 200ok<Entity-Body containing rewind response in any suitable media protocol>".

The HTTP protocol can be also used to send media data across firewalls. The client can send a GET request to the video server, and the video server can then send the video data as the Entity-Body of the HTTP response to this GET request.

Some firewalls may be restrictive with respect to HTTP data and may permit HTTP packets to traverse only on a certain port, e.g., port **80** and/or port **8080**. FIG. **5C** illustrates one such situation. In this case, the control and data communications for the various data stream, e.g., audio, video, and/or annotation associated with different rendering sessions (and different clients) may be multiplexed using conventional multiplexer code and/or circuit **506** at client application **300** prior to being sent via port **502** (which may represent, for example, HTTP port **80** or HTTP port **8080**). The inventive combined use of the HTTP protocol and of the multiplexer for transmitting media control and data is referred to as the HTTP multiplex protocol, and can be used to send this data across firewalls that only allow HTTP traffic on specific ports, e.g., port **80** or **8080**.

At server **402**, representing, for example, server **104** of FIG. **1**, conventional demultiplexer code and/or circuit **508** may be employed to decode the received data packets to identify which stream the control request is associated with. Likewise, data sent from server **402** to client application **300** may be multiplexed in advance at server **402** using for example conventional multiplexer code and/or circuit **510**. The multiplexed data is then sent via port **502**. At client application **300**, the multiplexed data may be decoded via conventional demultiplexer code and/or circuit **512** to identify which stream the received data packets is associated with (audio, video, or annotation).

Multiplexing and demultiplexing at the client and/or server may be facilitated for example by the use of the Request-URL part of the Request-Line of HTTP requests. As mentioned above, the structure of HTTP requests is described in RFC 1945. The Request-URL may, for example, identify the stream associated with the data and/or control request being transmitted. In one embodiment, the additional information in the Request-URL in the HTTP header may be as small as one or a few bits added to the LHTTP request sent from client application **300** to server **402**.

To further facilitate discussion of the inventive HTTP multiplexing technique, reference may now be made to FIG. 5D. In FIG. 5D, the plug-in application **660** within client plug-in application **660** may attempt to receive media data (e.g., video, audio, annotation, or the like) by first sending a control request **670** to server **664** via control connection **665**. The control request is an HTTP request, which arrives at the HTTP default port **654** on server **664**. As mentioned earlier, the default HTTP port may be either port **80** or port **8080** in one embodiment.

In one example, the control request **670** from client plug-in **660** takes the form of a command to "POST/12469

HTTP/1.0<Entity-Body>" which indicates to the server (through the designation 12469 as the Request-URL) that this is a control connection. The Entity-Body contains, as described above, binary data that informs the video server that the client plug-in **660** wants to display a certain video or audio clip. Software codes within server **664** may be employed to assign a unique ID to this particular request from this particular client.

For discussion sake, assume that server **664** associates unique ID 35,122 with a video data connection between itself and client plug-in application **660**, and unique ID 29,999 with an audio data connection between itself and client plug-in application. The unique ID is then communicated as message **672** from server **664** to client plug-in application **660**, again through the aforementioned HTTP default port using data connection **667**. The Entity-Body of message **672** contains, among other things and as depicted in detail **673**, the audio and/or video session ID. Note that the unique ID is unique to each data connection (e.g., each of the audio, video, and annotation connections) of each client plug-in application (since there may be multiple client plug-in applications attempting to communicate through the same port).

Once the connection is established, the same unique ID number is employed by the client to issue HTTP control requests to server **664**. By way of example, client plug-in application **660** may issue a command "GET/35,122 HTTP/1.0" or "POST/35,122 HTTP/1.0<Entity-Body containing binary data with the REWIND media command>" to request a video file or to rewind on the video file. Although the rewind command is used in FIGS. **5A–5D** to facilitate ease of discussion, other media commands, e.g., fast forward, pause, real-time play, live-play, or the like, may of course be sent in the Entity-Body. Note that the unique ID is employed in place of or in addition to the Request-URL to qualify the Request-URL.

Once the command is received by server **664**, the unique ID number (e.g. 35,122) may be employed by the server to demultiplex the command to associate the command with a particular client and data file. This unique ID number can also attach to the HTTP header of HTTP responses sent from server **664** to client plug-in application **660**, through the same HTTP default port **654** on server **664**, to permit client plug-in application **660** to ascertain whether an HTTP data packet is associated with a given data stream.

Advantageously, the invention permits media control commands and media data to be communicated between the client computer and the server computer via the default HTTP port, e.g., port **80** or **8080** in one embodiment, even if HTTP packets are otherwise blocked by a firewall disposed between the client computer and the server computer. The association of each control connection and data connection to each client with a unique ID advantageously permits multiple control and data connections (from one or more clients) to be established through the same default HTTP port on the server, advantageously bypassing the firewall. Since both the server and the client have the demultiplexer code and/or circuit that resolve a particular unique ID into a particular data stream, multiplexed data communication is advantageously facilitated thereby.

In some networks, it may not be possible to traverse the firewall due to stringent firewall policies. As mentioned earlier, it may be possible in these situations to allow the client application to communicate with a server using a proxy. FIG. **6** illustrates this situation wherein client application **300** employs proxy **602** to communicate with server **402**. The use of proxy **602** may be necessary since client application **300** may employ a protocol which is strictly prohibited by firewall **604**. The identity of proxy **602** may be found in browser program **306**, e.g., Netscape as it employs the proxy to download its web pages, or may be configured by the user himself. Typical protocols that may employ a proxy for communication, e.g., proxy **602**, includes HTTP and UDP.

In accordance with one embodiment of the present invention, the multiple protocols that may be employed for communication between a server computer and a client computer are tried in parallel during autodetect. In other words, the connections depicted in FIGS. **3**, **4**, **5**C, and **6** may be attempted simultaneously and in parallel over different control connections by the client computer. Via these control connections, the server is requested to respond with various protocols.

If the predefined "best" protocol (predetermined in accordance with some predefined protocol priority) is received by the client application from the server, autodetect may, in one embodiment, end immediately and the "best" protocol is selected for immediate communication. In one real-time data rendering application, UDP is considered the "best" protocol, and the receipt of UDP data by the client may trigger the termination of the autodetect.

If the "best" protocol has not been received after a predefined time period, the most advantageous protocol (in terms of for example data transfer rate and/or transmission control) is selected among the set of protocols received by the client. The selected protocol may then be employed for communication between the client and the server.

FIG. **7** depicts, in accordance with one embodiment of the present invention, a simplified flowchart illustrating the steps of the inventive autodetect technique. In FIG. **7**, the client application starts (in step **702**) by looking up the HTTP proxy, if there is any, from the browser. As stated earlier, the client computer may have received a web page from the browser, which implies that the HTTP protocol may have been employed by the browser program for communication. If a HTTP proxy is required, the name and location of the HTTP proxy is likely known to the browser, and this knowledge may be subsequently employed by the client to at least enable communication with the server using the HTTP proxy protocol, i.e., if a more advantageous protocol cannot be ascertained after autodetect.

In step **704**, the client begins the autodetect sequence by starting in parallel the control thread **794**, along with five protocol threads **790**, **792**, **796**, **798**, and **788**. As the term is used herein, parallel refers to both the situation wherein the multiple protocol threads are sent parallelly starting at substantially the same time (having substantially simnilar starting time), and the situation wherein the multiple protocol threads simultaneously execute (executing at the same time), irrespective when each protocol thread is initiated. In the latter case, the multiple threads may have, for example, staggered start time and the initiation of one thread may not depend on the termination of another thread.

Control tread **794** represents the thread for selecting the most advantageous protocol for communication. The other protocol threads **790**, **792**, **796**, **798**, and **788** represent threads for initiating in parallel communication using the various protocols, e.g., UDP, TCP, HTTP proxy, HTTP through port **80** (HTTP **80**), and HTTP through port **8080** (HTTP **8080**). Although only five protocol threads are shown, any number of protocol threads may be initiated by the client, using any conventional and/or suitable protocols.

The steps associated with each of threads **794**, **790**, **792**, **796**, **798**, and **788** are discussed herein in connection with FIGS. **8**A–**8**E and **9**.

In FIG. **8**A, the UDP protocol thread is executed. The client inquires in step **716** whether there requires a UDP proxy. If the UDP proxy is required, the user may obtain the name of the UDP proxy from, for example, the system administrator in order to use the UDP proxy to facilitate communication to the proxy (in step **718**). If no UDP proxy is required, the client may directly connect to the server (in step **720**). Thereafter, the client may begin sending a data request (i.e., a control request) to the server in step **722** using the UDP protocol (either through the proxy if a proxy is involved or directly to the server if no proxy is required).

In FIG. **8**B, the TCP protocol thread is executed. If TCP protocol is employed, the client typically directly connects to the server (in step **726**). Thereafter, the client may begin sending a data request (i.e., a control request) to the server using the TCP protocol (step **724**).

In FIG. **8**C, the HTTP protocol thread is executed. The client inquires in step **716** whether there requires a HTTP proxy. If the HTTP proxy is required, the user may obtain the name of the HTTP proxy from, for example, the browser since, as discussed earlier, the data pertaining to the proxy may be kept by the browser. Alternatively, the user may obtain data pertaining to the HTTP proxy from the system administrator in order to use the HTTP proxy to facilitate communication to the server (in step **732**).

If no HTTP proxy is required, the client may directly connect to the server (in step **730**). Thereafter, the client may begin sending a data request (i.e., a control request) to the server in step **734** using the HTTP protocol (either through the proxy if a proxy is involved or directly to the server if no proxy is required).

In FIG. **8**D, the HTTP **80** protocol thread is executed. If HTTP **80** protocol is employed, HTTP data may be exchanged but only through port **80**, which may be for example the port on the client computer through which communication with the network is permitted. Through port **80**, the client typically directly connects to the server (in step **736**). Thereafter, the client may begin sending a data request (i.e., a control request) to the server (step **738**) using the HTTP **80** protocol.

In FIG. **8**E, the HTTP **8080** protocol thread is executed. If HTTP **8080** protocol is employed, HTTP data may be exchanged but only through port **8080**, which may be the port on the client computer for communicating with the network. Through port **8080**, the client typically directly connects to the server (in step **740**). Thereafter, the client may begin sending a data request (i.e., a control request) to the server (step **742**) using the HTTP **8080** protocol. The multiplexing and demultiplexing techniques that may be employed for communication through port **8080**, as well as port **80** of FIG. **8**D, have been discussed earlier and are not repeated here for brevity sake.

FIG. **9** illustrates, in accordance with one embodiment of the present invention, control thread **794** of FIG. **7**. It should be emphasized that FIG. **7** is but one way of implementing the control thread; other techniques of implementing the control thread to facilitate autodetect should be apparent to those skilled in the art in view of this disclosure. In step **746**, the thread determines whether the predefined timeout period has expired. The predefined timeout period may be any predefined duration (such as 7 seconds for example) from the time the data request is sent out to the server (e.g., step **722** of FIG. **8**A). In one embodiment, each protocol thread has its own timeout period whose expiration occurs at the expiration of a predefined duration after the data request using that protocol has been sent out. When all the timeout periods associated with all the protocols have been accounted for, the timeout period for the autodetect technique is deemed expired.

If the timeout has occured, the thread moves to step **754** wherein the most advantageous protocol among the set of protocols received back from the server is selected for communication. As mentioned, the selection of the most advantageous protocol may be performed in accordance with some predefined priority scheme, and data regarding the selected protocol may be saved for future communication sessions between this server and this client.

If no timeout has occurred, the thread proceeds to step **748** to wait for either data from the server or the expiration of the timeout period. If timeout occurs, the thread moves to step **754**, which has been discussed earlier. If data is received from the server, the thread moves to step **750** to ascertain whether the protocol associated with the data received from the server is the predefined "best" protocol, e.g., in accordance with the predefined priority.

If the predefined "best" protocol (e.g., UDP in some real-time data rendering applications) is received, the thread preferably moves to step **754** to terminate the autodetect and to immediately begin using this protocol for data communication instead of waiting of the timeout expiration. Advantageously, the duration of the autodetect sequence may be substantially shorter than the predefined timeout period. In this manner, rapid autodetect of the most suitable protocol and rapid establishment of communication are advantageously facilitated.

If the predefined "best" protocol is not received in step **750**, the thread proceeds to step **752** to add the received protocol to the received set. This received protocol set represents the set of protocols from which the "most advantageous" (relatively speaking) protocol is selected. The most advantageous protocol is ascertained relative to other protocols in the received protocol set irrespective whether it is the predefined "best" protocol in accordance with the predefined priority. As an example of a predefined protocol priority, UDP may be deemed to be best (i.e., the predefined best), followed by TCP, HTTP, then HTTP **80** and HTTP **8080** (the last two may be equal in priority). As mentioned earlier, the most advantageous protocol is selected from the received protocol set preferably upon the expiration of the predefined timeout period.

From step **752**, the thread returns to step **746** to test whether the timeout period has expired. If not, the thread continues along the steps discussed earlier.

Note that since the invention attempts to establish communication between the client application and the server computer in parallel, the time lag between the time the autodetect mechanism begins to execute and the time when the most advantageous protocol is determined is minimal. If communication attempts have been tried in serial, for example, the user would suffer the delay associated with each protocol thread in series, thereby disadvantageously lengthening the time period between communication attempt and successful establishment of communication.

The saving in time is even more dramatic in the event the network is congested or damaged. In some networks, it may take anywhere from 30 to 90 seconds before the client application realizes that an attempt to connect to the server (e.g., step **720**, **726**, **730**, **736**, or **740**) has failed. If each protocol is tried in series, as is done in one embodiment, the delay may, in some cases, reach minutes before the user realizes that the network is unusable and attempts should be made at a later time.

By attempting to establish communication via the multiple protocols in parallel, network-related delays are suffered in parallel. Accordingly, the user does not have to wait for multiple attempts and failures before being able to ascertain that the network is unusable and an attempt to establish communication should be made at a later time. In one embodiment, once the user realizes that all parallel attempts to connect with the network and/or the proxies have failed, there is no need to make the user wait until the expiration of the timeout periods of each thread. In accordance with this embodiment, the user is advised to try again as soon as it is realized that parallel attempts to connect with the server have all failed. In this manner, less of the user's time is needed to establish optimal communication with a network.

While this invention has been described in terms of several preferred embodiments, there are alterations, permutations, and equivalents which fall within the scope of this invention. For example, although the invention has been described with reference with sending out protocol threads in parallel, the automatic protocol detection technique also applies when the protocol threads are sent serially. In this case, while it may take longer to select the most advantageous protocol for selection, the automatic protocol detection technique accomplishes the task without requiring any sophisticated technical knowledge on the part of the user of the client computer. The duration of the autodetect technique, even when serial autodetect is employed, may be shortened by trying the protocols in order of their desirability and ignoring less desirable protocols once a more desirable protocol is obtained. It should also be noted that there are many alternative ways of implementing the methods and apparatuses of the present invention. It is therefore intended that the following appended claims be interpreted as including all such alterations, permutations, and equivalents as fall within the true spirit and scope of the present invention.

What is claimed is:

1. A method for employing a Hypertext Transfer Protocol (HTTP protocol) for transmitting streamed digital media data from a server, said server being configured for coupling to a client via computer network, comprising:

receiving at said server from said client an HTTP POST request, said POST request requesting a first portion of said digital media data and including a request header and a request entity-body, said request entity body including a media command for causing said first portion of said digital media data to be sent from said server to said client; and

sending an HTTP response to said client from said server, said HTTP response including a response header and a response entity body, said response entity body including at least a portion of said first portion of said digital media data, wherein said digital media data includes video data.

2. A method for employing a Hypertext Transfer Protocol (IITTP protocol) for transmitting streamed digital media data from a server, said server being configured for coupling to a client via computer network, comprising:

receiving at said server from said client an HTTP POST request, said POST request requesting a first portion of said digital media data and including a request header and a request entity-body, said request entity body including a media command for causing said first portion of said digital media data to be sent from said server to said client; and sending an HTTP response to said client from said server, said HTTP response including a response header and a response entity body, said response entity body including at least a portion of said first portion of said digital media data,

wherein said digital media data represents live video data, said live video data representing data to be rendered at said client as an event related to said live video data is being recorded.

3. The method of claim **2** wherein said media command includes a REWIND command and said first portion of said digital media data represents digital media data that is earlier in time than a video frame currently displayed at said client.

4. The method of claim **2** wherein said media command includes a FAST FORWARD command and said first portion of said digital media data represents digital media data that is later in time than a video frame currently displayed at said client.

5. The method of claim **2** wherein said media command includes a LIVE PLAY command and said first portion of said digital media data represents digital media data that is recorded live.

6. The method of claim **2** wherein said media command includes a REAL-TIME PLAY command and said first portion of said digital media data represents digital media data that is stored earlier and streamed through said server.

7. The method of claim **2** wherein said HTTP protocol is permitted on only a selected port of said server, said HTTP port request and said HTTP response being multiplexed through said selected port.

8. The method of claim **7** wherein said selected port represents one of a port **80** and a port **8080** on said server.

9. The method of claim **8** wherein said server is employed to receive a plurality of HTTP requests from other clients coupled to said server, said plurality of HTTP requests also being multiplexed via said selected port.

10. The method of claim **9** wherein said HTTP requests includes a unique ID, said unique ID identifying said digital media data as digital media requested by said client, said unique ID being assigned by said server when said client establishes connection with said server.

11. A computer readable medium containing readable instructions for transmitting streamed media data employing a Hypertext Transfer Protocol (HTTP protocol) form a server, said server being configured for coupling to a client via a computer network, said computer readable instructions implementing the steps of:

receiving at said server from said client an HTTP POST request, said POST request requesting a first portion of said digital media data and including a request header and a request entity-body, said request entity body including a media command for causing said first portion of said digital media data to be sent from said server to said client; and

sending an IITTP response to said client from said server, said HTTP response including a response header and a response entity body, said response entity body including at least a portion of said digital media data, wherein said digital media data includes video data.

12. A computer readable medium containing readable instructions for transmitting streamed media data employing a Hypertext Transfer Protocol (HTTP protocol) form a server, said server being configured for coupling to a client via a computer network, said computer readable instructions implementing the steps of:

receiving at said server from said client an HTTP POST request, said POST request requesting a first portion of said digital media data and including a request header and a request entity-body, said request entity body including a media command for causing said first portion of said digital media data to be sent from said server to said client; and

sending an HTTP response to said client from said server, said HTTP response including a response header and a response entity body, said response entity body including at least a portion of said digital media data,

wherein said digital media data represents live video data, said live video data representing data to be rendered at said client as an event related to said live video data is being recorded.

13. The computer readable medium of claim **12** wherein said media command includes a REWIND command and said first portion of said digital media data represents digital media data that is earlier in time than a video frame currently displayed at said client.

14. The computer readable medium of claim **12** wherein said media command includes a FAST FORWARD command and said first portion of said digital media data represents digital media data that is later in time than a video frame currently displayed at said client.

15. The computer readable medium of claim **12** wherein said media command includes a LIVE PLAY command and said first portion of said digital media data represents digital media data that is recorded live.

16. The computer readable medium of claim **12** wherein said media command includes a REAL-TIME PLAY command and said first portion of said digital media data represents digital media data that is stored earlier and streamed through said server.

17. The computer readable medium of claim **12** wherein said HTTP protocol is permitted on only a selected port of said server, said HTTP port request and said HTTP response being multiplexed through said selected port.

18. The computer readable medium of claim **17** wherein said selected port represents one of a port **80** and a port **8080** on said server.

19. The computer readable medium of claim **18** wherein said server is employed to receive a plurality of HTTP requests from other clients coupled to said server, said plurality of HTTP requests also being multiplexed via said selected port.

20. The computer readable medium of claim **19** wherein said HTTP requests includes a unique ID, said unique ID identifying said digital media data as digital media requested by said client, said unique ID being assigned by said server when said client establishes connection with said server.

21. A method for employing a Hypertext Transfer Protocol (HTTP protocol) for transmitting streamed digital media data from a server, said server being configured for coupling to a client via computer network, comprising:

receiving at said server from said client an HTTP POST request, said POST request requesting a first portion of said digital media data and including a request header and a request entity-body, said request entity body including a media command for causing said first portion of said digital media data to be sent from said server to said client; and

sending an HTTP response to said client from said server, said HTTP response including a response header and a response entity body said response entity body including at least a portion of said first portion of said digital media data, wherein the digital media data comprises audio data.

22. A computer readable medium containing readable instructions for transmitting streamed media data employing a Hypertext Transfer Protocol (HTTP protocol) form a server, said server being configured for coupling to a client via a computer network, said computer readable instructions implementing the steps of:

receiving at said server from said client an HTTP POST request, said POST request requesting a first portion of said digital media data and including a request header and a request entity-body, said request entity body including a media command for causing said first portion of said digital media data to be sent from said server to said client; and

sending an HTTP response to said client from said server, said HTTP response including a response header and a response entity body said response entity body including at least a portion of said digital media data, wherein the digital media data comprises audio data.

23. A system comprising:

a client sending a Hypertext Transfer Protocol (HTTP protocol) POST request requesting a first portion of a digital media data and including a media command for causing the first portion of the digital media data to be sent from the server to the client; and

a server coupled to the client, wherein the server is sending an HTTP response to the client, wherein the HTTP response has at least a portion of the first portion of the digital media data, wherein said digital media data includes video data.

24. A system comprising:

a client sending a Hypertext Transfer Protocol (HTTP protocol) POST request requesting a first portion of a digital media data and including a media command for causing the first portion of the digital media data to be sent from the server to the client; and

a server coupled to the client, wherein the server is sending an HTTP response to the client, wherein the HTTP response has at least a portion of the first portion of the digital media data,

wherein the digital media data comprises a live video data, the live video data representing data to be rendered at the client as an event related to the live video data is being recorded.

25. The system of claim **24** wherein the HTTP POST request comprises a media command for causing the first portion of the digital media data to be sent from the server to the client.

26. A system comprising:

a client sending a Hypertext Transfer Protocol (HTTP protocol) POST request requesting a first portion of a digital media data and including a media command for causing the first portion of the disital media data to be sent from the server to the client; and

a server coupled to the client, wherein the server is sending an HTTP response to the client, wherein the HTTP response has at least a portion of the first portion of the digital media data, wherein the HTTP POST request comprises a media command for causing the first portion of the digital media data to be sent from the server to the client,

wherein the media command includes a REWIND command and wherein the first portion of the digital media data represents digital media data that is earlier in time than a video frame currently displayed at the client.

27. The system of claim **25** wherein the media command includes a FAST FORWARD command and wherein the first portion of the digital media data represents digital media data that is later in time than a video frame currently displayed at the client.

28. The system of claim **25** wherein the media command includes a LIVE PLAY command and wherein the first portion of the digital media data represents digital media data that is recorded live.

29. The system of claim **25** wherein the media command includes a REAL-TIME PLAY command and wherein the first portion of the digital media data represents digital media data that is stored earlier and streamed through the server.

30. The system of claim **26** wherein a HTTP protocol is permitted on only a selected port of the server, the HTTP protocol request and the HTTP response being multiplexed through the selected port.

31. The system of claim **30** wherein the selected port represents one of a port **80** and a port **8080** on the server.

32. The system of claim **31** wherein the server is employed to receive a number of HTTP requests from other clients coupled to the server, the number of IITTP requests also being multiplexed via the selected port.

33. The system of claim **32** wherein the number of HTTP requests comprises a unique identification (ID) identifying the digital media data as digital media requested by the client, the unique ID being assigned by the server when the client establishes connection with the server.

34. A system comprising:

a client sending a Hypertext Transfer Protocol (HTTP protocol) POST request requesting a first portion of a digital media data and including a media command for causing the first portion of the digital media data to be sent from the server to the client; and

a server coupled to the client, wherein the server is sending an HTTP response to the client, wherein the HTTP response has at least a portion of the first portion of the digital media data,

wherein the digital media data comprises video data.

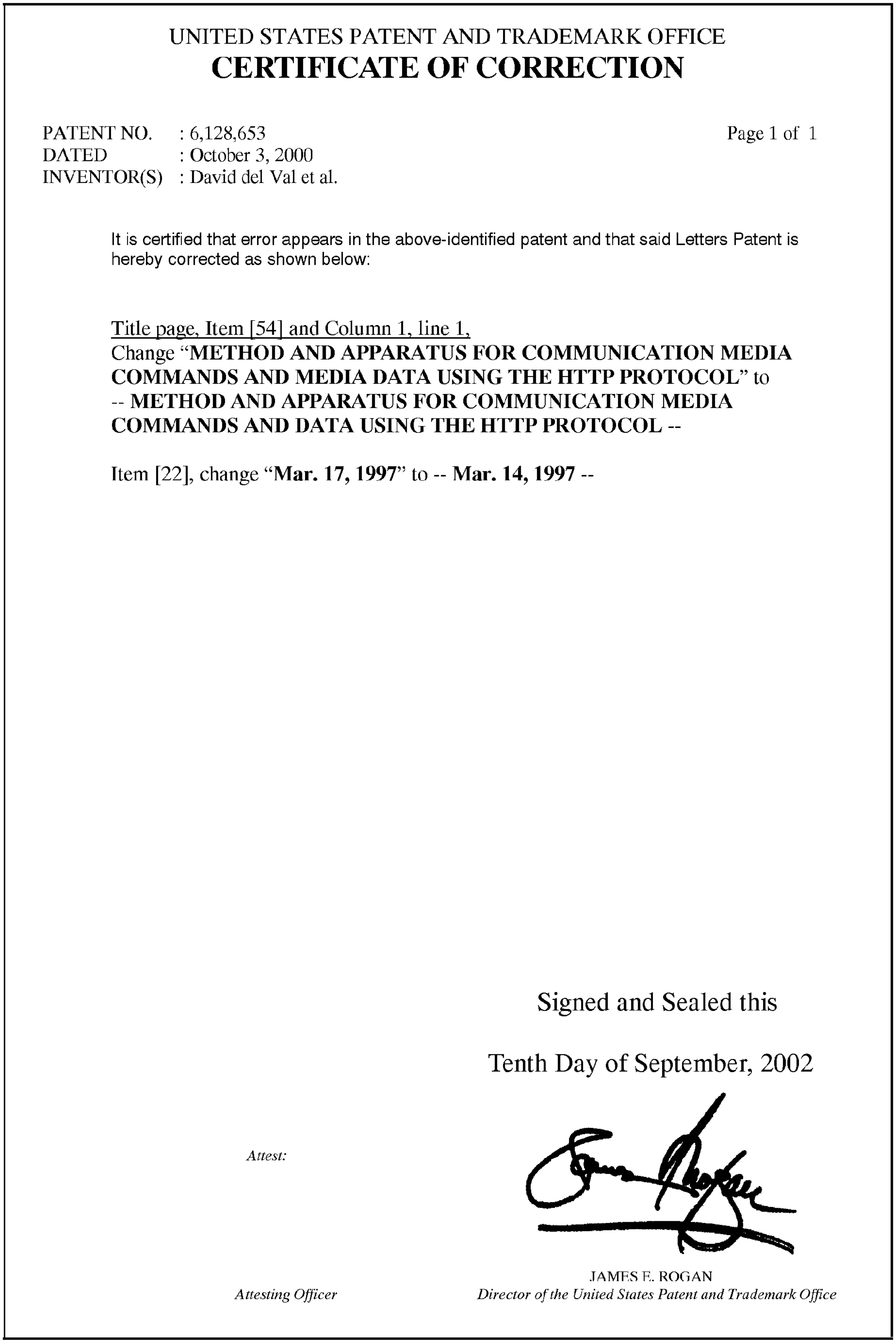
35. A system comprising:

a client sending a Hypertext Transfer Protocol (HTTP protocol) POST request requesting a first portion of a digital media data and including a media command for causing the first portion of the digital media data to be sent from the server to the client; and

a server coupled to the client, wherein the server is sending an HTTP response to the client, wherein the HTTP response has at least a portion of the first portion of the digital media data,

wherein the digital media data comprises audio data.

* * * * *



UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 6,128,653 Page 1 of 1
DATED : October 3, 2000
INVENTOR(S) : David del Val et al.

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

Title page, Item [54] and Column 1, line 1,
Change "**METHOD AND APPARATUS FOR COMMUNICATION MEDIA COMMANDS AND MEDIA DATA USING THE HTTP PROTOCOL**" to -- **METHOD AND APPARATUS FOR COMMUNICATION MEDIA COMMANDS AND DATA USING THE HTTP PROTOCOL** --

Item [22], change "**Mar. 17, 1997**" to -- **Mar. 14, 1997** --

Signed and Sealed this

Tenth Day of September, 2002

Attest:

JAMES E. ROGAN
Director of the United States Patent and Trademark Office

Attesting Officer

US006795848B1

(12) **United States Patent**
Border et al.

(10) **Patent No.: US 6,795,848 B1**
(45) **Date of Patent: Sep. 21, 2004**

(54) **SYSTEM AND METHOD OF READING AHEAD OF OBJECTS FOR DELIVERY TO AN HTTP PROXY SERVER**

(75) Inventors: **John Border**, Germantown, MD (US); **Douglas Dillon**, Gaithersburg, MD (US); **Matt Butehorn**, Mt. Airy, MD (US)

(73) Assignee: **Hughes Electronics Corporation**, El Segundo, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 765 days.

(21) Appl. No.: **09/708,134**

(22) Filed: **Nov. 8, 2000**

(51) **Int. Cl.**^{7} **G06F 15/167**
(52) **U.S. Cl.** **709/213**; 709/203; 709/201
(58) **Field of Search** 709/201–203, 709/238, 225, 226, 229, 213; 711/117, 118, 122, 137

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,995,725 A	*	11/1999	Dillon	709/203
6,112,228 A	*	8/2000	Earl et al.	709/205
6,226,635 B1	*	5/2001	Katariya	707/4
6,282,542 B1	*	8/2001	Carneal et al.	707/10
6,442,651 B2	*	8/2002	Crow et al.	711/118
6,658,463 B1		12/2003	Dillon et al.	709/219

FOREIGN PATENT DOCUMENTS

DE	198 21 876 A	11/1999
WO	WO 98/53410	11/1998
WO	WO 99/08429	2/1999

OTHER PUBLICATIONS

Luotonen A: "Web Proxy Servers", Web Proxy Servers, XX, XX, 1998 pp. 156–170, XP002928629 p. 170, line 12–p. 170, line 40.

Ari Loutonen, Web Proxy Servers, XP002928629, pp. 156–170, Prentice Hall PTR, Upper Saddle River, NJ.

H. Inoue, et al., "An Adaptive WWW Cache Mechanism in the AI3 Network", *Proceedings of the INET'97*, www.isoc.org/inet97/proceedings/A1/A1_2.HTM, pp. 1–5.

Y. Zhang, et al., "HBX High Bandwidth X for Satellite Internetworking", 10th Annual X Technical Conference, San Jose, CA, Feb. 12–14, 1996 (The X Resource, Issue 17, pp. 85–94), pp. 1–10.

T. Baba, et al., "AI^3 Satellite Internet Infrastructure and the Deployment in Asia", IEICE Trans. Commun., vol. E84–B, No. 8, Aug. 2001, pp. 2048–2057.

H. Inoue, "An Adaptive WWW Cache Mechanism in the AI3 Network", INET'97 (Jun. 24–27, 1997), www.ai3.net/pub/inet97/cache ppt/foils.html, pp. 1–9.

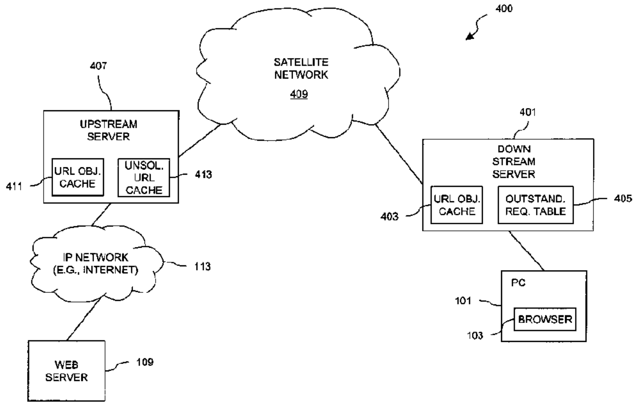
* cited by examiner

Primary Examiner—Mehmet B. Geckil
(74) *Attorney, Agent, or Firm*—John T. Whelan

(57) **ABSTRACT**

A communication system for retrieving web content is disclosed. A downstream proxy server receives a URL request message from a web browser, in which the URL request message specifies a URL content that has an embedded object. An upstream proxy server receives the URL request message from the downstream proxy server. The upstream proxy server selectively forwards the URL request message to a web server and receives the URL content from the web server, wherein the upstream proxy server forwards the URL content to the downstream proxy server and parses the URL content to obtain the embedded object prior to receiving a corresponding embedded object request message initiated by the web browser.

4 Claims, 7 Drawing Sheets



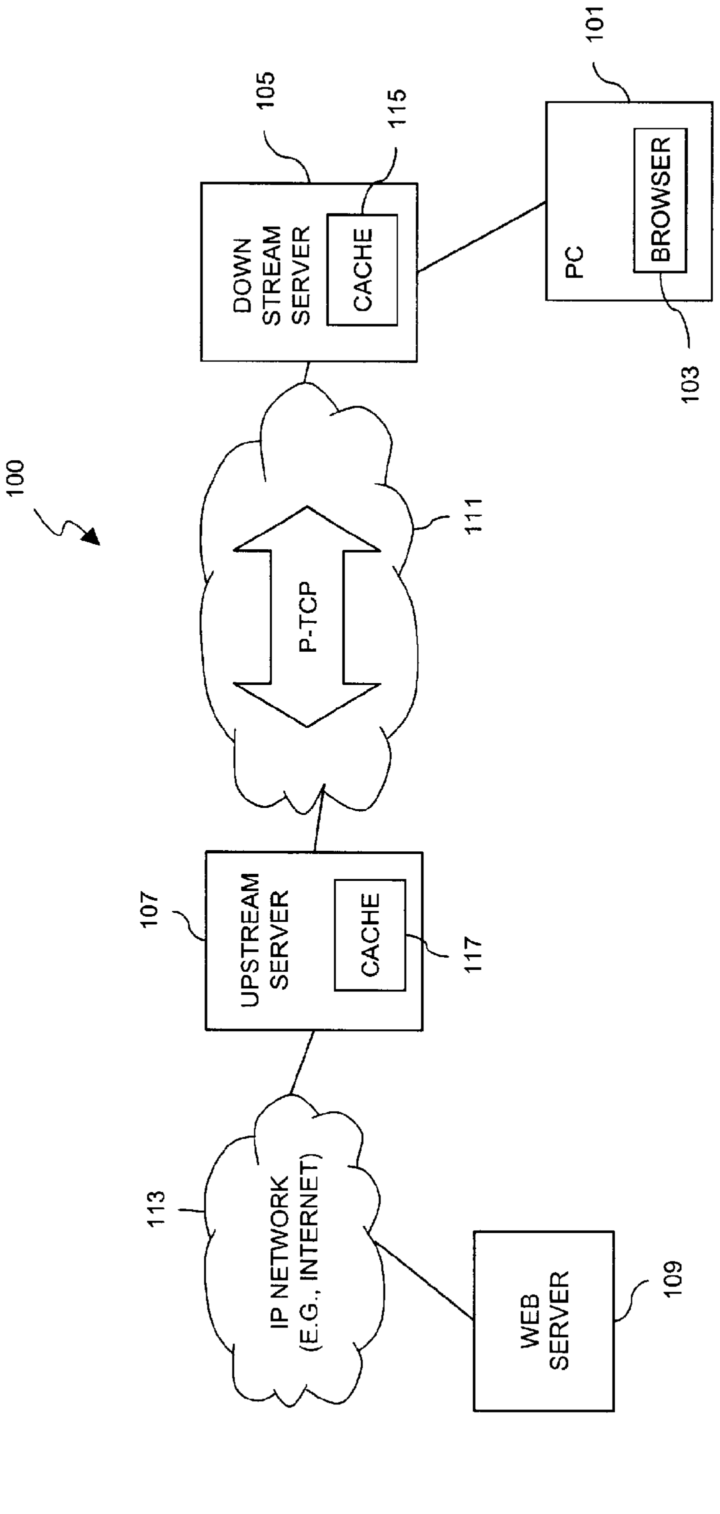

100
WEB SERVER
109
IP NETWORK (E.G., INTERNET)
113
UPSTREAM SERVER
107
CACHE
117
P-TCP
111
DOWN STREAM SERVER
105
CACHE
115
PC
101
BROWSER
103

FIG. 1

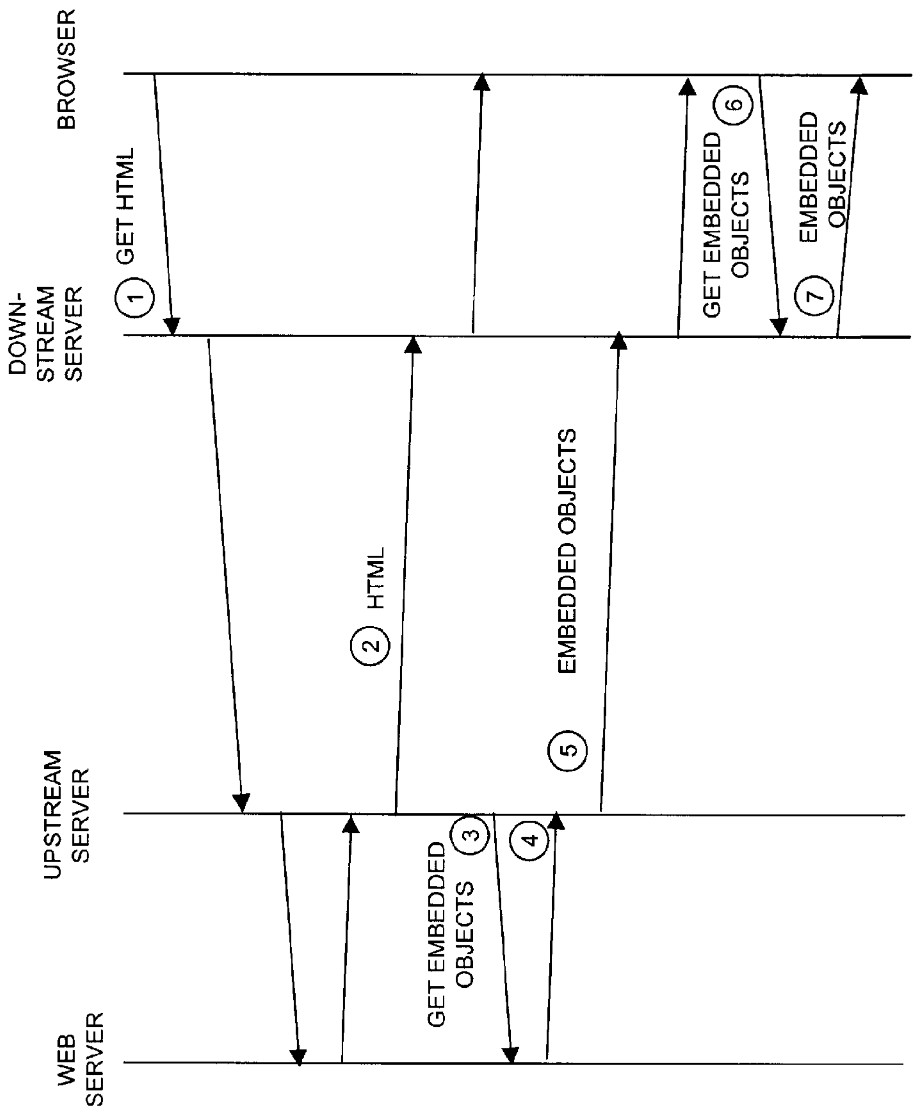
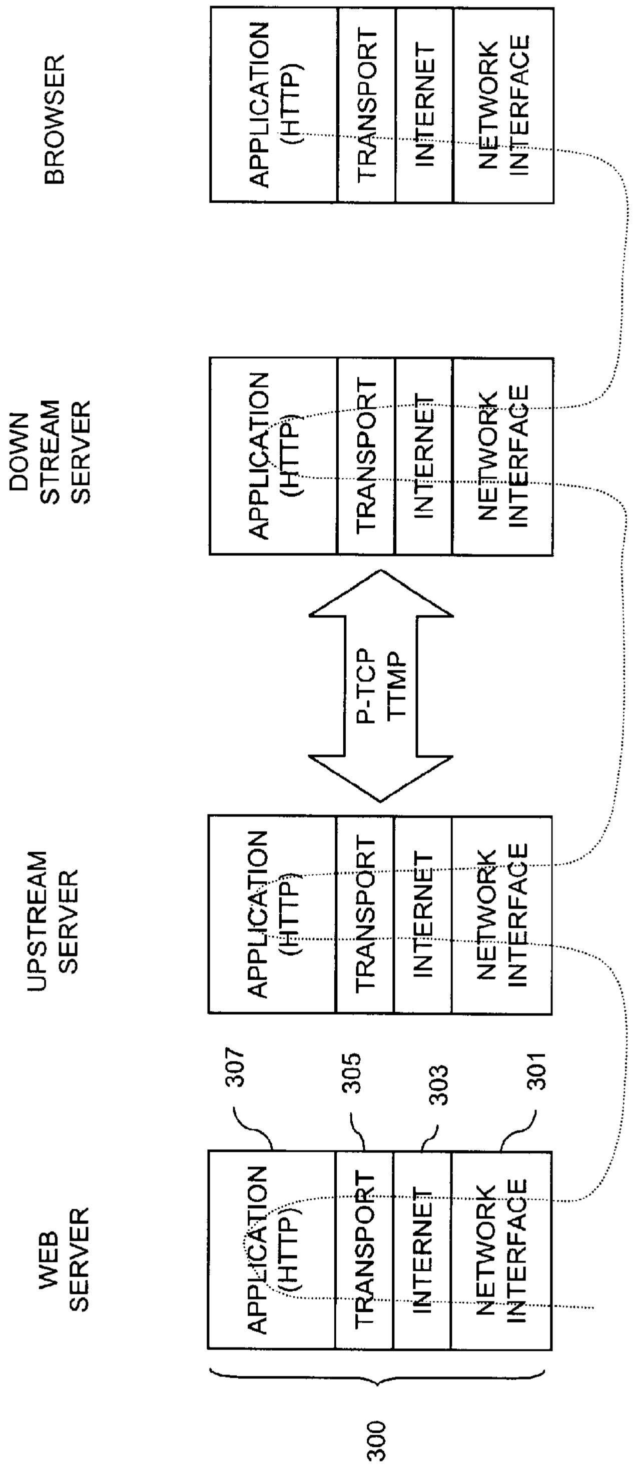

BROWSER
DOWN-
STREAM
SERVER
UPSTREAM
SERVER
WEB
SERVER
1
GET HTML
2
HTML
3
GET EMBEDDED
OBJECTS
4
5
EMBEDDED OBJECTS
6
GET EMBEDDED
OBJECTS
7
EMBEDDED
OBJECTS

FIG. 2

FIG. 3


WEB SERVER
UPSTREAM SERVER
DOWN STREAM SERVER
BROWSER
APPLICATION (HTTP)
TRANSPORT
INTERNET
NETWORK INTERFACE
APPLICATION (HTTP)
TRANSPORT
INTERNET
NETWORK INTERFACE
APPLICATION (HTTP)
TRANSPORT
INTERNET
NETWORK INTERFACE
APPLICATION (HTTP)
TRANSPORT
INTERNET
NETWORK INTERFACE
P-TCP
TTMP
307
305
303
301
300

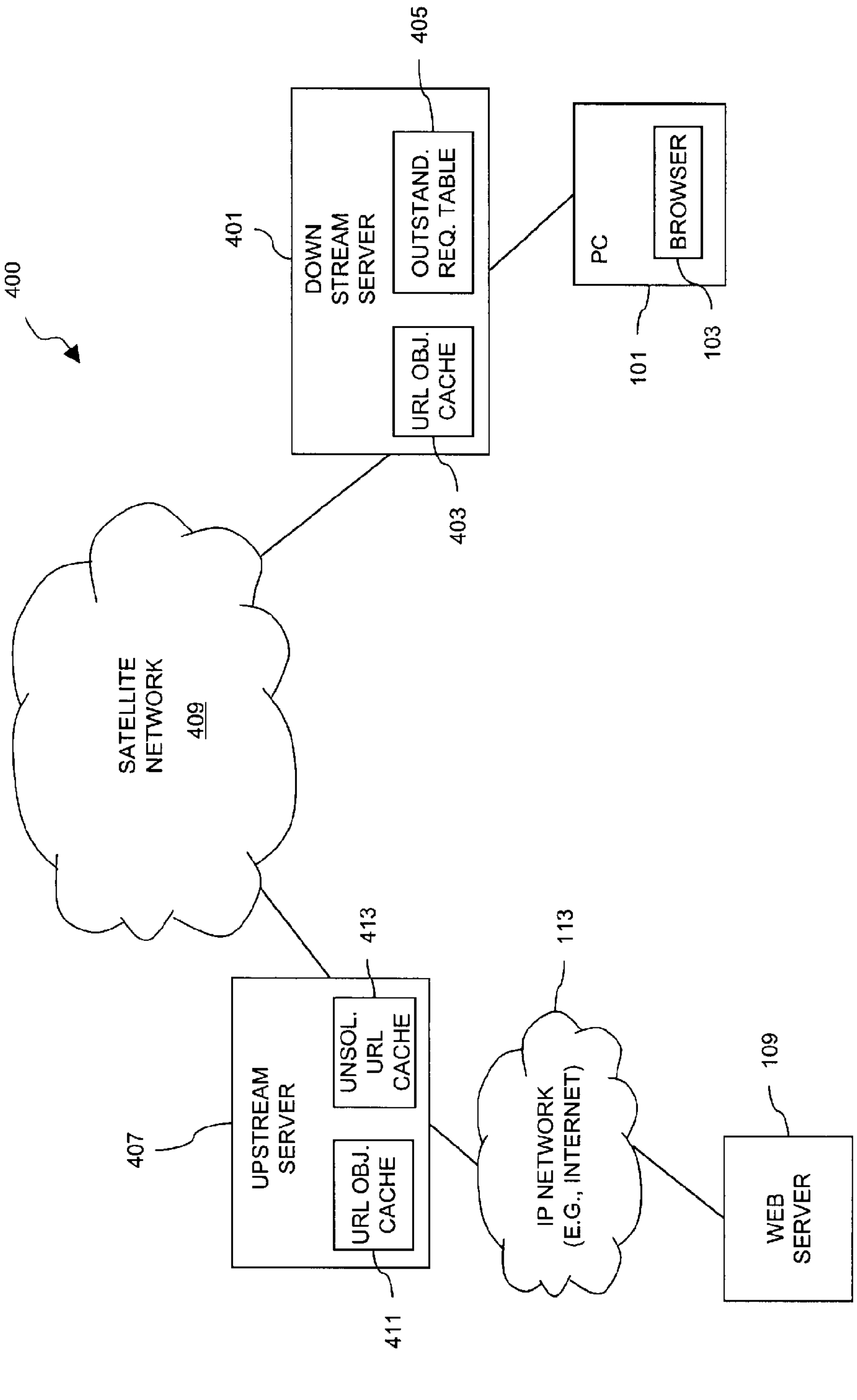

400
401
DOWN STREAM SERVER
URL OBJ. CACHE
403
OUTSTAND. REQ. TABLE
405
PC
BROWSER
101
103
SATELLITE NETWORK
409
UPSTREAM SERVER
407
URL OBJ. CACHE
411
UNSOL. URL CACHE
413
IP NETWORK (E.G., INTERNET)
113
WEB SERVER
109

FIG. 4

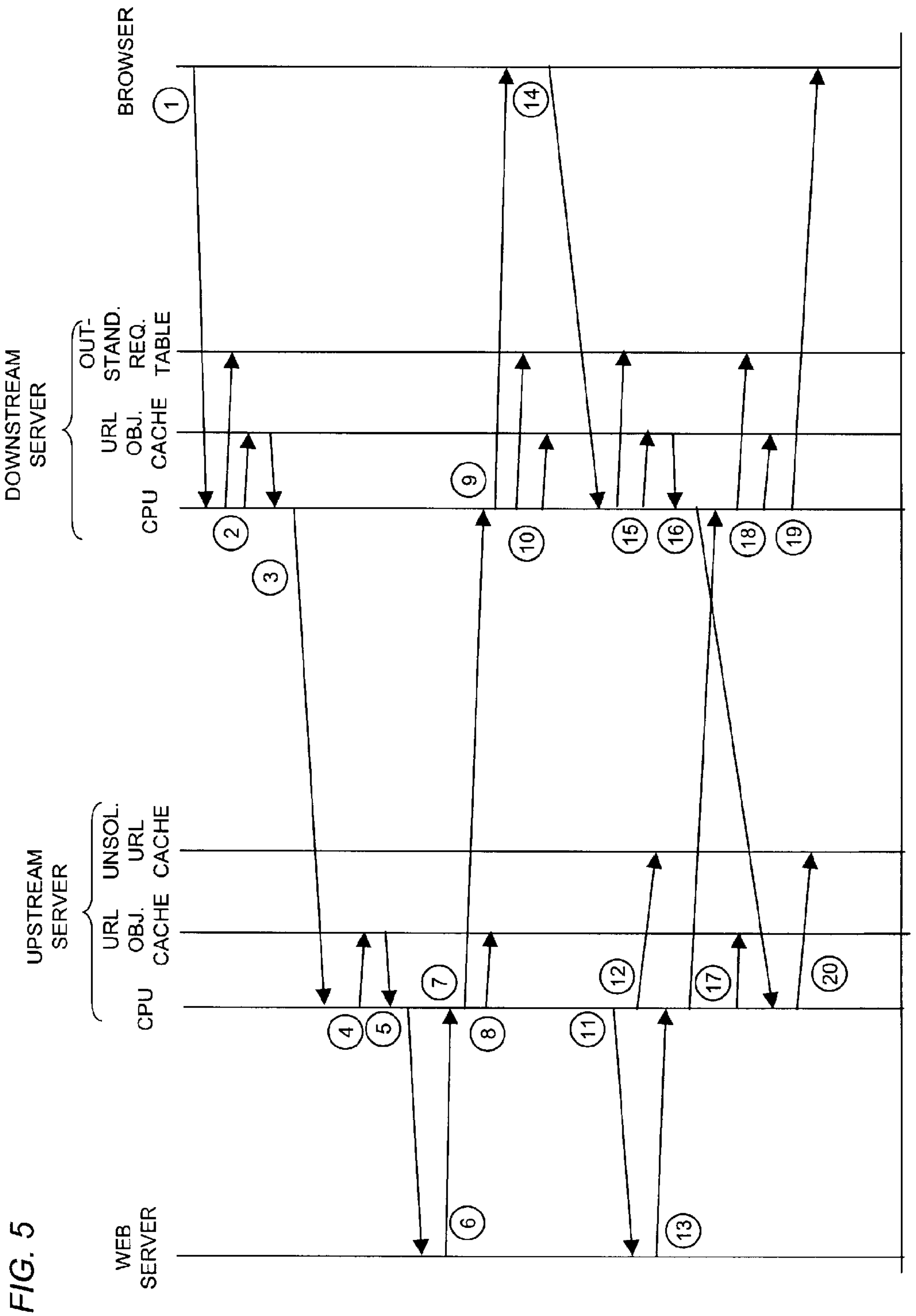

BROWSER
DOWNSTREAM SERVER
CPU
URL OBJ. CACHE
OUT-STAND. REQ. TABLE
UPSTREAM SERVER
CPU
URL OBJ. CACHE
UNSOL. URL CACHE
WEB SERVER
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20

FIG. 5

PRIOR ART

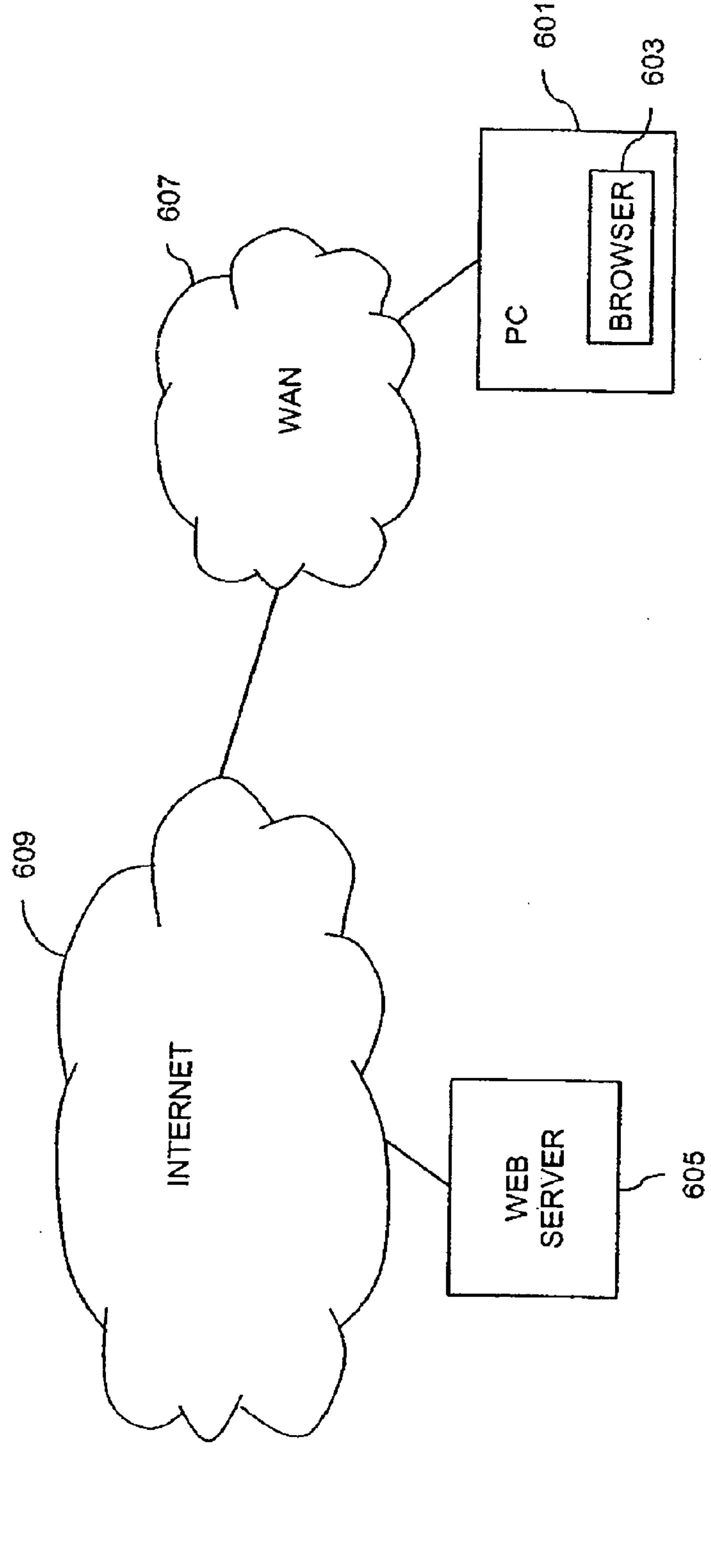


FIG. 6

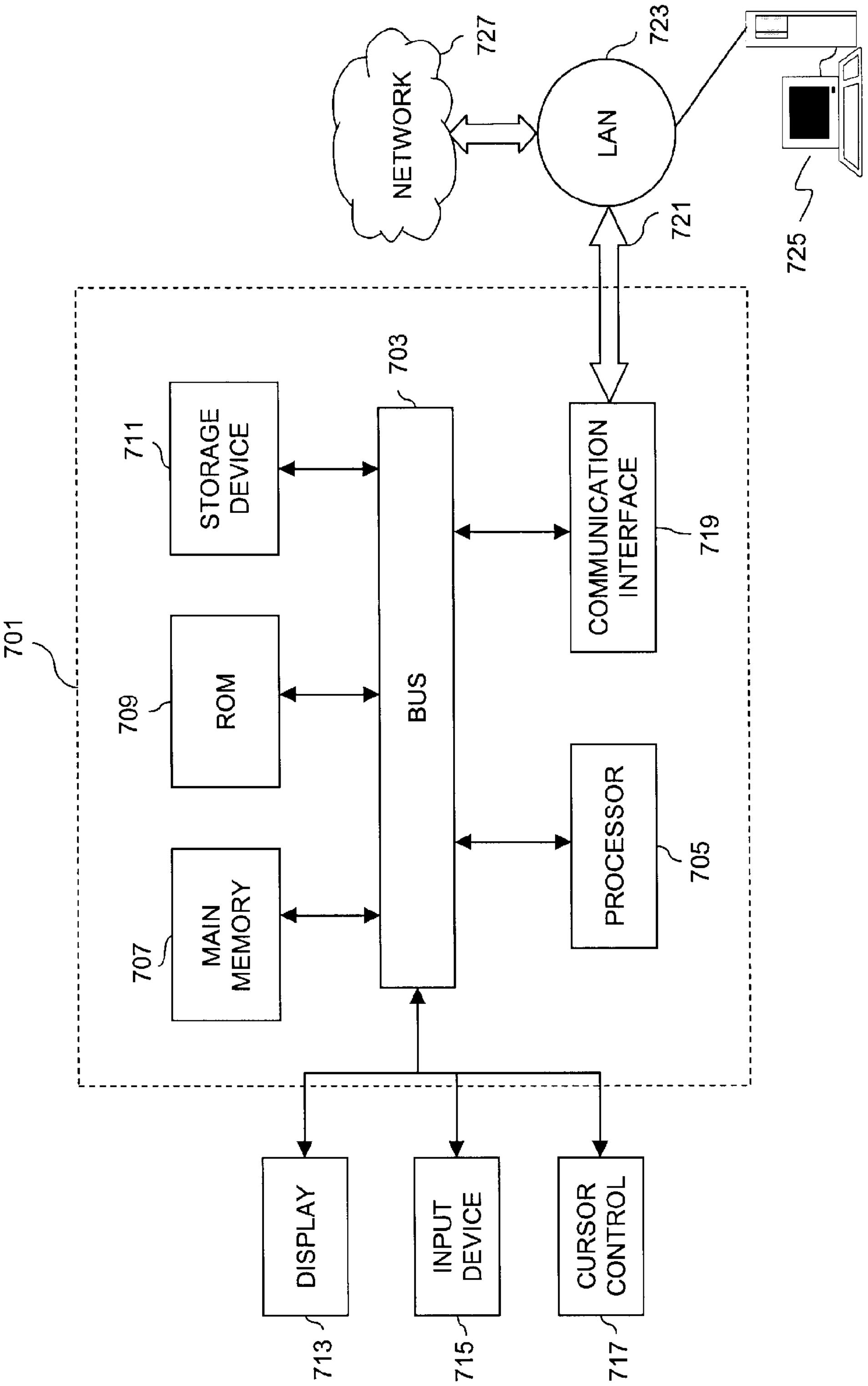

701
MAIN MEMORY
707
ROM
709
STORAGE DEVICE
711
BUS
703
PROCESSOR
705
COMMUNICATION INTERFACE
719
721
LAN
723
NETWORK
727
725
DISPLAY
713
INPUT DEVICE
715
CURSOR CONTROL
717

FIG. 7

SYSTEM AND METHOD OF READING AHEAD OF OBJECTS FOR DELIVERY TO AN HTTP PROXY SERVER

CROSS-REFERENCES TO RELATED APPLICATION

This application is related to co-pending U.S. patent application Ser. No. 09/498,936, filed Feb. 4, 2000, entitled "Satellite Multicast Performance Enhancing Multicast HTTP Proxy System and Method," the entirety of which is incorporated herein by reference.

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention relates to a communication system, and is more particularly related to retrieving web content using proxy servers.

2. Discussion of the Background

As businesses and society, in general, become increasingly reliant on communication networks to conduct a variety of activities, ranging from business transactions to personal entertainment, these communication networks continue to experience greater and greater delay, stemming in part from traffic congestion and network latency. For example, the maturity of electronic commerce and acceptance of the Internet, in particular the World Wide Web ("Web"), as a daily tool pose an enormous challenge to communication engineers to develop techniques to reduce network latency and user response times. With the advances in processing power of desktop computers, the average user has grown accustomed to sophisticated applications (e.g., streaming video, radio broadcasts, video games, etc.), which place tremendous strain on network resources. The Web as well as other Internet services rely on protocols and networking architectures that offer great flexibility and robustness; however, such infrastructure may be inefficient in transporting Web traffic, which can result in large user response time, particularly if the traffic has to traverse an intermediary network with a relatively large latency (e.g., a satellite network).

FIG. **6** is a diagram of a conventional communication system for providing retrieval of web content by a personal computer (PC). PC **601** is loaded with a web browser **603** to access the web pages that are resident on web server **605**; collectively the web pages and web server **605** denote a "web site." PC **603** connects to a wide area network (WAN) **607**, which is linked to the Internet **609**. The above arrangement is typical of a business environment, whereby the PC **601** is networked to the Internet **609**. A residential user, in contrast, normally has a dial-up connection (not shown) to the Internet **609** for access to the Web. The phenomenal growth of the Web is attributable to the ease and standardized manner of "creating" a web page, which can possess textual, audio, and video content.

Web pages are formatted according to the Hypertext Markup Language (HTML) standard which provides for the display of high-quality text (including control over the location, size, color and font for the text), the display of graphics within the page and the "linking" from one page to another, possibly stored on a different web server. Each HTML document, graphic image, video clip or other individual piece of content is identified, that is, addressed, by an Internet address, referred to as a Uniform Resource Locator (URL). As used herein, a "URL" may refer to an address of an individual piece of web content (HTML document, image, sound-clip, video-clip, etc.) or the individual piece of content addressed by the URL. When a distinction is required, the term "URL address" refers to the URL itself while the terms "web content", "URL content" or "URL object" refers to the content addressed by the URL.

In a typical transaction, the user enters or specifies a URL to the web browser **603**, which in turn requests a URL from the web server **605**. The web server **605** returns an HTML page, which contains numerous embedded objects (i.e., web content), to the web browser **603**. Upon receiving the HTML page, the web browser **603** parses the page to retrieve each embedded object. The retrieval process often requires the establishment of separate communication sessions (e.g., TCP (Transmission Control Protocol) sessions) to the web server **605**. That is, after an embedded object is received, the TCP session is torn down and another TCP session is established for the next object. Given the richness of the content of web pages, it is not uncommon for a web page to possess over 30 embedded objects. This arrangement disadvantageously consumes network resources, but more significantly, introduces delay to the user.

Delay is further increased if the WAN **607** is a satellite network, as the network latency of the satellite network is conventionally a longer latency than terrestrial networks. In addition, because HTTP utilizes a separate TCP connection for each transaction, the large number of transactions amplifies the network latency. Further, the manner in which frames are created and images are embedded in HTML requires a separate HTTP transaction for every frame and URL compounds the delay.

Based on the foregoing, there is a clear need for improved approaches for retrieval of web content within a communication system.

There is a need to utilize standard protocols to avoid development costs and provide rapid industry acceptance.

There is also a need for a web content retrieval mechanism that makes the networks with relatively large latency viable and/or competitive for Internet access.

Therefore, an approach for retrieving web content that reduces user response times is highly desirable.

SUMMARY OF THE INVENTION

According to one aspect of the invention, a communication system for retrieving web content comprises a downstream proxy server that is configured to receive a URL request message from a web browser. The URL request message specifies a URL content that has an embedded object. An upstream proxy server is configured to communicate with the downstream proxy server and to receive the URL request message from the downstream proxy server. The upstream proxy server selectively forwards the URL request message to a web server and receives the URL content from the web server, wherein the upstream proxy server forwards the URL content to the downstream proxy server and parses the URL content to obtain the embedded object prior to the web browser having to issue an embedded object request message. The above arrangement advantageously reduces user response time associated with web browsing.

BRIEF DESCRIPTION OF THE DRAWINGS

A more complete appreciation of the invention and many of the attendant advantages thereof will be readily obtained as the same becomes better understood by reference to the following detailed description when considered in connection with the accompanying drawings, wherein:

FIG. 1 is a diagram of a communication system employing a downstream proxy server and an upstream proxy server for accessing a web server, according to an embodiment of the present invention;

FIG. 2 is a sequence diagram of the process of reading ahead used in the system of FIG. 1;

FIG. 3 is a block diagram of the protocols utilized in the system of FIG. 1;

FIG. 4 is a diagram of a communication system employing a downstream proxy server and an upstream proxy server that maintains an unsolicited URL (Uniform Resource Locator) cache for accessing a web server, according to an embodiment of the present invention;

FIG. 5 is a sequence diagram of the process of reading ahead used in the system of FIG. 4;

FIG. 6 is a diagram of a conventional communication system for providing retrieval of web content by a personal computer (PC); and

FIG. 7 is a diagram of a computer system that can be configured as a proxy server, in accordance with an embodiment of the present invention.

DESCRIPTION OF THE PREFERRED EMBODIMENTS

In the following description, for the purpose of explanation, specific details are set forth in order to provide a thorough understanding of the invention. However, it will be apparent that the invention may be practiced without these specific details. In some instances, well-known structures and devices are depicted in block diagram form in order to avoid unnecessarily obscuring the invention.

The present invention provides a communication system for retrieving web content. A downstream proxy server receives a URL request message from a web browser, in which the URL request message specifies a URL content that has an embedded object. An upstream proxy server receives the URL request message from the downstream proxy server. The upstream proxy server selectively forwards the URL request message to a web server and receives the URL content from the web server. The upstream proxy server forwards the URL content to the downstream proxy server and parses the URL content to obtain the embedded object prior to receiving a corresponding embedded object request message initiated by the web browser.

Although the present invention is discussed with respect to a protocols and interfaces to support communication with the Internet, the present invention has applicability to any protocols and interfaces to support a packet switched network, in general.

FIG. 1 shows a diagram of a communication system employing a downstream proxy server and an upstream proxy server for accessing a web server, according to an embodiment of the present invention. Communication system **100** includes a user station **101** that utilizes a standard web browser **103** (e.g., Microsoft Internet Explorer, Netscape Navigator). In this example, the user station **101** is a personal computer (PC); however, any computing platform may be utilized, such as a workstation, web enabled set-top boxes, web appliances, etc. System **100** utilizes two proxy servers **105** and **107**, which are referred to as a downstream proxy server **105** and an upstream proxy server **107**, respectively. PC **101** connects to downstream server **105**, which communicates with upstream server **107** through a network **111**. This communication with downstream server **105** may be transparent to PC **101**. According to an embodiment of the present invention, the network **111** is a VSAT (Very Small Aperture Terminal) satellite network. Alternatively, the network **111** may be any type of Wide Area Network (WAN); e.g., ATM (Asynchronous Transfer Mode) network, router-based network, T1 network, etc. The upstream server **107** has connectivity to an IP network **113**, such as the Internet, to access web server **109**.

Proxy servers **105** and **107**, according to an embodiment of the present invention, are Hypertext Transfer Protocol (HTTP) proxy servers with HTTP caches **115** and **117**, respectively. These servers **105** and **107** communicate using persistent connections (which is a feature of HTTP 1.1). Use of persistent connections enables a single TCP connection to be reused for multiple requests of the embedded objects within a web page associated with web server **109**. Further, TCP Transaction Multiplexing Protocol (TTMP) may be utilized. TTMP and persistent-TCP are more fully described with respect to FIG. **3**.

Web browser **103** may be configured to either access URLs directly from a web server **109** or from HTTP proxy servers **105** and **107**. A web page may refer to various source documents by indicating the associated URLs. As discussed above, a URL specifies an address of an "object" in the Internet **113** by explicitly indicating the method of accessing the resource. A representative format of a URL is as follows: http://www.hns.com/homepage/document.html. This example indicates that the file "document.html" is accessed using HTTP.

HTTP proxy server **105** and **107** acts as an intermediary between one or more browsers and many web servers (e.g., server **109**). A web browser **103** requests a URL from the proxy server (e.g., **105**) which in turn "gets" the URL from the addressed web server **109**. An HTTP proxy **105** itself may be configured to either access URLs directly from a web server **109** or from another HTTP proxy server **107**.

According to one embodiment of the present invention, the proxy servers **105** and **107** may support multicast delivery. IP multicasting can be used to transmit information from upstream server **107** to multiple downstream servers (of which only one downstream server **105** is shown). A multicast receiver (e.g., a network interface card (NIC)) for the downstream proxy server **105** operates in one of two modes: active and inactive. In the active mode operation, the downstream proxy server **105** opens multicast addresses and actively processes the received URLs on those addresses. During the inactive mode, the downstream proxy server **105** disables multicast reception from the upstream proxy server **107**. In the inactive state the downstream proxy server **105** minimizes its use of resources by, for example, closing the cache and freeing its RAM memory (not shown).

For downstream proxy server **105** operating on a general purpose personal computer, the multicast receiver for the downstream proxy server **105** may be configured to switch between the active and inactive states to minimize the proxy server's interfering with user-directed processing. The downstream proxy server **105** utilizes an activity monitor which monitors user input (key clicks and mouse clicks) to determine when it should reduce resource utilization. The downstream proxy server **105** also monitors for proxy cache lookups to determine when it should go active.

Upon boot up, the multicast receiver is inactive. After a certain amount of time with no user interaction and no proxy cache lookups (e.g., 10 minutes), the downstream proxy server **105** sets the multicast receiver active. The downstream proxy server **105** sets the multicast receiver active immediately upon needing to perform a cache lookup. The downstream proxy server **105** sets the multicast receiver inactive whenever user activity is detected and the cache **115** has not had any lookups for a configurable period of time (e.g., 5 minutes).

For downstream proxy servers **105** running on systems with adequate CPU (central processing unit) resources to simultaneously handle URL reception and other applications, the user may configure the downstream proxy server **105** to set the multicast receiver to stay active regardless of user activity. The operation of system **100** in the retrieval of web content, according to an embodiment of the present invention, is described in FIG. **2**, below.

FIG. **2** shows a sequence diagram of the process of reading ahead used in the system of FIG. **1**. To retrieve a web page (i.e., HTML page) from web server **109**, the web browser **103** on PC **101** issues an HTTP GET request, which is received by down stream proxy server **105** (per step **1**). For the purposes of explanation, the HTML page is addressed as URL "HTML." The downstream server **105** checks its cache **115** to determine whether the requested URL has been previously visited. If the downstream proxy server **105** does not have URL HTML stored in cache **115**, the server **105** relays this request, GET URL "HTML", to upstream server **117**.

The HTTP protocol also supports a GET IF MODIFIED SINCE request wherein a web server (or a proxy server) either responds with a status code indicating that the URL has not changed or with the URL content if the URL has changed since the requested date and time. This mechanism updates cache **115** of proxy server **105** only if the contents have changed, thereby saving unnecessary processing costs.

The upstream server **117** in turn searches for the URL HTML in its cache **117**; if the HTML page is not found in cache **117**, the server **117** issues the GET URL HTML request the web server **109** for the HTML page. Next, in step **2**, the web server **109** transmits the requested HTML page to the upstream server **117**, which stores the received HTML page in cache **117**. The upstream server **117** forwards the HTML page to the down stream server **105**, and ultimately to the web browser **103**. The HTML page is stored in cache **115** of the downstream server **105** as well as the web browser's cache (not shown). In step **3**, the upstream server **117** parses the HTML page and requests the embedded objects within the HTML page from the web server **109**; the embedded objects are requested prior to receiving corresponding embedded object requests initiated by the web browser **103**.

Step **3** may involve the issuance of multiple GET requests; the web page within web server **109** may contain over 30 embedded objects, thus requiring 30 GET requests. In effect, this scheme provides a way to "read ahead" (i.e., retrieve the embedded object) in anticipation of corresponding requests by the web browser **103**. The determination to read-ahead may be based upon explicit tracking of the content of the downstream server cache **115**; only those embedded objects that are not found in the cache **115** are requested. Alternatively, the upstream server **107** may only request those embedded objects that are not in the upstream server cache **117**. Further, in actual implementation wherein multiple web servers exist, the upstream server **107** may track which web server tend to transmit uncachable objects; for such servers, objects stored therein are read-ahead.

Moreover if the HTML contains a cookie and the GET HTML request is directed to the same web server, then the upstream server **107** includes the cookie in the read-ahead request to the web server **109** for the embedded objects. A cookie is information that a web server **109** stores on the client system, e.g., PC **101**, to identify the client system. Cookies provide a way for the web server **109** to return customized web pages to the PC **101**.

In step **4**, the web server **109** honors the GET request by transmitting the embedded objects to the upstream server **107**. The upstream server **107**, as in step **5**, then forwards the retrieved objects to the downstream server **105**, where the objects are stored until they are requested by the web browser **103**. It should be noted that the upstream server **107** forwards the embedded objects prior to being requested to do so by the web browser **103**; however, the upstream server **107** performs this forwarding step based on an established criteria. There are scenarios in which all the embedded objects that are read-ahead may not subsequently be requested by the web browser **103**. In such cases, if the upstream server **107** transfers these embedded objects over network **111** to the downstream server **105**, the bandwidth of network **111** would be wasted, along with the resources of the downstream server **105**. Accordingly, the forwarding criteria need to reflect the trade off between response time and bandwidth utilization. These forwarding criteria may include the following: (1) object size, and (2) "cachability." That is, upstream server **107** may only forward objects that are of a predetermined size or less, so that large objects (which occupy greater bandwidth) are not sent to the downstream server **105**. Additionally, if the embedded object is marked uncacheable, then the object may be forwarded to the downstream server **105**, which by definition will not have the object stored. The upstream server **107** may be configured to forward every retrieved embedded object, if bandwidth is not a major concern.

In the scenario in which the embedded objects correspond to a request that contains a cookie, the upstream server **107** provides an indication whether the embedded objects has the corresponding cookie.

In step **6**, the web browser **103** issues a GET request for the embedded objects corresponding to the web page within the web server **109**. The downstream server **105** recognizes that the requested embedded objects are stored within its cache **115** and forwards the embedded objects to the web browser **103**. Under this approach, the delays associated with network **111** and the Internet **113** are advantageously avoided.

The caching HTTP proxy servers **105** and **107**, according to one embodiment of the present invention, stores the most frequently accessed URLs. When web server **109** delivers a URL to the proxy servers **105** and **107**, the web server **109** may deliver along with the URL an indication of whether the URL should not be cached and an indication of when the URL was last modified.

At this point, web browser **103** has already requested URL HTML, and has the URL HTML stored in a cache (not shown) of the PC **101**. To avoid stale information, the web browser **103** needs to determine whether the information stored at URL HTML has been updated since the time it was last requested. As a result, the browser **103** issues a GET HTML IF MODIFIED SINCE the last time HTML was obtained. Assuming that URL HTML was obtained at 11:30 a.m. on Sep. 22, 2000, browser **103** issues a GET HTML IF MODIFIED SINCE Sep. 22, 2000 at 11:30 a.m. request. This request is sent to downstream proxy server **105**. If downstream proxy server **105** has received an updated version of URL HTML since Sep. 22, 2000 at 11:30 a.m., downstream proxy server **105** supplies the new URL HTML information to the browser **103**. If not, the downstream proxy server **105** issues a GET IF MODIFIED SINCE command to upstream proxy server **107**. If upstream proxy server **107** has received an updated URL HTML since Sep. 22, 2000 at 11:30 a.m., upstream proxy server **107** passes the new URL HTML to the downstream proxy server **105**. If not, the upstream proxy server **107** issues a GET HTML IF MODIFIED SINCE command to the web server **109**. If URL HTML has not changed since Sep. 22, 2000 at 11:30 a.m., web server **109** issues a NO CHANGE response to the upstream proxy server **107**. Under this arrangement, bandwidth and processing time are saved, since if the URL HTML has not been modified since the last request, the entire contents of URL HTML need not be transferred between web browser **103**, downstream proxy server **105**, upstream proxy server **107**, and the web server **109**, only an indication that there has been no change need be exchanged. Caching proxy servers **105** and **107** offer both reduced network utilization and reduced response time when they are able to satisfy requests with cached URLs.

FIG. **3** shows a block diagram of the protocols utilized in the system of FIG. **1**. The servers **105**, **107**, and **109** and PC **101** employ, according to one embodiment of the present invention, a layered protocol stack **300**. The protocol stack **300** includes a network interface layer **301**, an Internet layer **303**, a transport layer **305**, and an application layer **307**.

HTTP is an application level protocol that is employed for information transfer over the Web. RFC (Request for Comment) **2616** specifies this protocol and is incorporated herein in its entirety. In addition, a more detailed definition of URL can be found in RFC **1737**, which is incorporated herein in its entirety.

The Internet layer **303** may be the Internet Protocol (IP) version 4 or 6, for instance. The transport layer **305** may include the TCP (Transmission Control Protocol) and the UDP (User Datagram Protocol). According to one embodiment of the present invention, at the transport layer, persistent TCP connections are utilized in the system **100**; in addition, TCP Transaction Multiplexing Protocol (TTMP) may be used.

The TCP Transaction Multiplexing Protocol (TTMP) allows multiple transactions, in this case HTTP transactions, to be multiplexed onto one TCP connection. Thus, transaction multiplexing provides an improvement over separate connection for each transaction (HTTP 1.0) and pipelining (HTTP 1.1) by preventing a single stalled request from stalling other requests. This is particularly beneficial when the downstream proxy server **105** is supporting simultaneous requests from multiple browsers (of which only browser **103** is shown in FIG. **1**).

The downstream proxy server **105** initiates and maintains a TCP connection to the upstream proxy server **107** as needed to carry HTTP transactions. The TCP connection could be set up and kept connected as long as the downstream proxy server **105** is running and connected to the network **111**. The persistent TCP connection may also be set up when the first transaction is required and torn down after the connection has been idle for some period.

An HTTP transaction begins with a request header, optionally followed by request content which is sent from the downstream proxy server **105** to the upstream proxy server **107**. An HTTP transaction concludes with a response header, optionally followed by response content. The downstream proxy server **105** maintains a transaction ID sequence number, which is incremented with each transaction. The downstream proxy server **105** breaks the transaction request into one or more blocks, creates a TTMP header for each block, and sends the blocks with a TTMP header to the upstream proxy server **107**. The upstream proxy server **107** similarly breaks a transaction response into blocks and sends the blocks with a TTMP header to the downstream proxy server **105**. The TTMP header contains the information necessary for the upstream proxy server **107** to reassemble a complete transaction command and to return the matching transaction response.

In particular, the TTMP header contains the following fields: a transaction ID field, a Block Length field, a Last Indication field, an Abort Indication field, and a Compression Information field. The transaction ID (i.e., the transaction sequence number) must rollover less frequently than the maximum number of supported outstanding transactions. The Block Length field allows a proxy server **105** and **107** to determine the beginning and ending of each block. The Last Indication field allows the proxy server **105** and **107** to determine when the end of a transaction response has been received. The Abort Indication field allows the proxy server **105** and **107** to abort a transaction when the transaction request or response cannot be completed. Lastly, the Compression Information field defines how to decompress the block.

The use of a single HTTP connection reduces the number of TCP acknowledgements that are sent over the network **111**. Reduction in the number of TCP acknowledgements significantly reduces the use of inbound networking resources which is particularly important when the network **111** is a VSAT system or other wireless systems. This reduction of acknowledgements is more significant when techniques, such as those described in U.S. Pat. No. 5,995,725 to Dillon entitled "Method and Apparatus for Requesting and Retrieving Information for a Source Computer Using Terrestrial and Satellite Interface" issued Nov. 30, 1999 (which is incorporated herein in its entirety), minimize the number of TCP acknowledgements per second per TCP connection.

Alternatively, downstream proxy server **105**, for efficiency, may use the User Datagram Protocol (UDP) to transmit HTTP GET and GET IF MODIFIED SINCE requests to the upstream proxy server **107**. This is done by placing the HTTP request header into the UDP payload. The use of UDP is very efficient as the overhead of establishing, maintaining and clearing TCP connections is not incurred. It is "best effort" in that there is no guarantee that the UDP packets will be delivered.

FIG. **4** shows a diagram of a communication system employing a downstream proxy server and an upstream proxy server that maintains an unsolicited URL (Uniform Resource Locator) cache for accessing a web server, according to an embodiment of the present invention. Communication system **400** employs a downstream server **401** that utilizes a cache **403** to store URL objects (i.e., embedded objects) as well as an Outstanding Request table **405**. The table **405** tracks the URL requests that the downstream server **401** has forwarded to upstream server **407**. In an embodiment of the present invention, the downstream server **401** and the upstream server **407** communicate over a satellite network **409**. The upstream server **407** maintains a URL object cache **411** for storing the embedded objects that are retrieved from web server **109**. In addition, the upstream server **407** uses an unsolicited URL cache **413**, which stores the URL requests for embedded objects in advance of the web browser **103** initiating such requests. The above arrangement advantageously enhances system performance.

FIG. **5** is a sequence diagram of the process of reading ahead used in the system of FIG. **4**. In step **1**, the web browser **101** sends a GET request (e.g., GET x.html) to the downstream server **401**. The downstream server **401** checks the URL object cache **403** to determine whether x.html is stored in the URL object cache **403**; if the content is stored in cache **403**, the downstream server **401** forwards the content to the browser **103**. Otherwise, the downstream server **401** writes the request in the Outstanding Request table **405** and sends the GET request to the upstream server **407** (step **3**). In this case, the web browser **103** and the downstream server **401** have not encountered the requested html page before. However, in the event that the web browser **103** has requested this HTML in the past or the downstream server **401** has stored this HTML previously, the latest time stamp is passed to the upstream server as a conditional GET request (e.g., GET IF MODIFIED SINCE Sep. 22, 2000). In this manner, only content that is more updated than the time stamp are retrieved. In step **4**, the upstream server **407** checks the URL object cache **411** in response to the received GET x.html request. Assuming x.html is not found in the URL object cache **411**, the upstream server **407** forwards the GET x.html request to the web server **109**, per step **5**. Accordingly, the web server **109**, as in step **6**, returns the web page to the upstream server **407**. In turn, the upstream server **407** forwards the web page to the downstream server **401**, as in step **7**, and stores the web page in the URL object cache **411**, per step **8**. In step **9**, the downstream server **401** sends the received web page to the web browser **103**. At this time, the downstream server **401** deletes the corresponding entry in the Outstanding Request table **405** and stores the received web page in the URL object cache **411** (step **10**).

Concurrent with steps **9** and **10**, the upstream server **407** parses the web page. The upstream server **407** then makes a determination, as in step **11**, to read-ahead the embedded objects of the web page based upon the read-ahead criteria that were discussed with respect to FIG. **2**, using a series of GET embedded object requests. Consequently, the upstream server **407** stores the URL x.html in the unsolicited URL cache **413**, per step **12**. In step **13**, the web server **109** returns the embedded objects to the upstream server **407**.

In step **14**, the web browser **103** parses the x.html page and issues a series of GET embedded objects requests. However, for explanatory purposes, FIG. **5** shows a single transaction for step **14**. In step **15**, the downstream server **401** checks its URL object cache **403** for the requested embedded object, and, assuming the particular object is not stored in cache **403**, writes an entry in the Outstanding Request table **405** corresponding to the GET embedded object request. Next, the downstream server **401** forwards the GET embedded object request to the upstream server **407** (per step **16**).

As shown in FIG. **5**, prior to the upstream server **407** receiving the GET embedded object request from the downstream server **401**, the upstream server **407** forwards the embedded objects to the downstream server **401** based on a forwarding criteria (as previously discussed with respect to FIG. **2**), storing these embedded objects in the URL object cache **413**. In step **18**, the downstream server **401** updates the Outstanding Request table **405** by deleting the GET embedded object request from the web browser **103**, and stores the received embedded object that has been read-ahead from the upstream server **407**. The embedded object is then transferred to the web browser **103** (per step **19**). In step **20**, upon receiving the GET embedded object from the downstream server **401**, the upstream server **407** discards the corresponding URL in the Unsolicited URL cache **413**. Under the above approach, the effects of network latencies associated with satellite network **409** and the Internet **113** are minimized, in that the web browser **103** receives the requested embedded object without having to wait for the full processing and transmission time associated with its GET embedded object request.

FIG. **7** is a diagram of a computer system that can be configured as a proxy server, in accordance with an embodiment of the present invention. Computer system **701** includes a bus **703** or other communication mechanism for communicating information, and a processor **705** coupled with bus **703** for processing the information. Computer system **701** also includes a main memory **707**, such as a random access memory (RAM) or other dynamic storage device, coupled to bus **703** for storing information and instructions to be executed by processor **705**. In addition, main memory **707** may be used for storing temporary variables or other intermediate information during execution of instructions to be executed by processor **705**. Computer system **701** further includes a read only memory (ROM) **709** or other static storage device coupled to bus **703** for storing static information and instructions for processor **705**. A storage device **711**, such as a magnetic disk or optical disk, is provided and coupled to bus **703** for storing information and instructions.

Computer system **701** may be coupled via bus **703** to a display **713**, such as a cathode ray tube (CRT), for displaying information to a computer user. An input device **715**, including alphanumeric and other keys, is coupled to bus **703** for communicating information and command selections to processor **705**. Another type of user input device is cursor control **717**, such as a mouse, a trackball, or cursor direction keys for communicating direction information and command selections to processor **705** and for controlling cursor movement on display **713**.

According to one embodiment, interaction within system **100** is provided by computer system **701** in response to processor **705** executing one or more sequences of one or more instructions contained in main memory **707**. Such instructions may be read into main memory **707** from another computer-readable medium, such as storage device **711**. Execution of the sequences of instructions contained in main memory **707** causes processor **705** to perform the process steps described herein. One or more processors in a multi-processing arrangement may also be employed to execute the sequences of instructions contained in main memory **707**. In alternative embodiments, hard-wired circuitry may be used in place of or in combination with software instructions. Thus, embodiments are not limited to any specific combination of hardware circuitry and software.

Further, the instructions to support the system interfaces and protocols of system **100** may reside on a computer-readable medium. The term "computer-readable medium" as used herein refers to any medium that participates in providing instructions to processor **705** for execution. Such a medium may take many forms, including but not limited to, non-volatile media, volatile media, and transmission media. Non-volatile media includes, for example, optical or magnetic disks, such as storage device **711**. Volatile media includes dynamic memory, such as main memory **707**. Transmission media includes coaxial cables, copper wire and fiber optics, including the wires that comprise bus **703**. Transmission media can also take the form of acoustic or light waves, such as those generated during radio wave and infrared data communication.

Common forms of computer-readable media include, for example, a floppy disk, a flexible disk, hard disk, magnetic tape, or any other magnetic medium, a CD-ROM, any other optical medium, punch cards, paper tape, any other physical medium with patterns of holes, a RAM, a PROM, and EPROM, a FLASH-EPROM, any other memory chip or cartridge, a carrier wave as described hereinafter, or any other medium from which a computer can read.

Various forms of computer readable media may be involved in carrying one or more sequences of one or more instructions to processor **705** for execution. For example, the instructions may initially be carried on a magnetic disk of a remote computer. The remote computer can load the instructions relating to the issuance of read-ahead requests remotely into its dynamic memory and send the instructions over a telephone line using a modem. A modem local to computer system **701** can receive the data on the telephone line and use an infrared transmitter to convert the data to an infrared signal. An infrared detector coupled to bus **703** can receive the data carried in the infrared signal and place the data on bus **703**. Bus **703** carries the data to main memory **707**, from which processor **705** retrieves and executes the instructions. The instructions received by main memory **707** may optionally be stored on storage device **711** either before or after execution by processor **705**.

Computer system **701** also includes a communication interface **719** coupled to bus **703**. Communication interface **719** provides a two-way data communication coupling to a network link **721** that is connected to a local network **723**. For example, communication interface **719** may be a network interface card to attach to any packet switched local area network (LAN). As another example, communication interface **719** may be an asymmetrical digital subscriber line (ADSL) card, an integrated services digital network (ISDN) card or a modem to provide a data communication connection to a corresponding type of telephone line. Wireless links may also be implemented. In any such implementation, communication interface **719** sends and receives electrical, electromagnetic or optical signals that carry digital data streams representing various types of information.

Network link **721** typically provides data communication through one or more networks to other data devices. For example, network link **721** may provide a connection through local network **723** to a host computer **725** or to data equipment operated by a service provider, which provides data communication services through a communication network **727** (e.g., the Internet). LAN **723** and network **727** both use electrical, electromagnetic or optical signals that carry digital data streams. The signals through the various networks and the signals on network link **721** and through communication interface **719**, which carry the digital data to and from computer system **701**, are exemplary forms of carrier waves transporting the information. Computer system **701** can transmit notifications and receive data, including program code, through the network(s), network link **721** and communication interface **719**.

The techniques described herein provide several advantages over prior approaches to retrieving web pages. A downstream proxy server is configured to receive a URL request message from a web browser, wherein the URL request message specifies a URL content that has an embedded object. An upstream proxy server is configured to communicate with the downstream proxy server and to receive the URL request message from the downstream proxy server. The upstream proxy server selectively forwards the URL request message to a web server and receives the URL content from the web server, wherein the upstream proxy server forwards the URL content to the downstream proxy server and parses the URL content to obtain the embedded object prior to receiving a corresponding embedded object request message initiated by the web browser. This approach advantageously improves user response time.

Obviously, numerous modifications and variations of the present invention are possible in light of the above teachings. It is therefore to be understood that within the scope of the appended claims, the invention may be practiced otherwise than as specifically described herein.

What is claimed is:

1. A communication system for retrieving web content, comprising:

a downstream proxy server configured to receive a URL request message from a web browser, the URL request message specifying a URL content having an embedded object; and

an upstream proxy server configured to communicate with the downstream proxy server and to receive the URL request message from the downstream proxy server, the upstream proxy server selectively forwarding the URL request message to a web server and receiving the URL content from the web server, wherein the upstream proxy server forwards the URL content to the downstream proxy server and parses the URL content to obtain the embedded object prior to receiving a corresponding embedded object request message initiated by the web browser,

wherein the upstream proxy server decides whether or not to send the embedded obiect to the downstream proxy server in accordance with the size of the embedded object.

2. A communication system for retrieving web content, comprising:

a downstream proxy server configured to receive a URL request message from a web browser, the URL request message specifying a URL content having an embedded object; and

an upstream proxy server configured to communicate with the downstream proxy server and to receive the URL request message from the downstream proxy server, the upstream proxy server selectively forwarding the URL request message to a web server and receiving the URL content from the web server, wherein the upstream proxy server forwards the URL content to the downstream proxy server and parses the URL content to obtain the embedded object prior to receiving a corresponding embedded object request message initiated by the web browser,

wherein the upstream proxy server decides whether or not to send the embedded object to the downstream proxy server in accordance with cachability of the embedded object.

3. A communication system for retrieving web content, comprising:

a downstream proxy server configured to receive a URL request message from a web browser, the URL request message specifying a URL content having an embedded object; and

an upstream proxy server configured to communicate with the downstream proxy server and to receive the URL request message from the downstream proxy server, the upstream proxy server selectively forwarding the URL request message to a web server and receiving the URL content from the web server, wherein the upstream proxy server forwards the URL content to the downstream proxy server and parses the URL content to obtain the embedded object prior to receiving a corresponding embedded object request message initiated by the web browser,

wherein in the case that the URL content has a cookie, the upstream proxy server uses the cookie when obtaining the embedded object.

4. A communication system for retrieving web content, comprising:

a downstream proxy server configured to receive a URL request message from a web browser, the URL request message specifying a URL content having an embedded object; and

an upstream proxy server configured to communicate with the downstream proxy server and to receive the URL request message from the downstream proxy server, the upstream proxy server selectively forwarding the URL request message to a web server and receiving the URL content from the web server, wherein the upstream proxy server forwards the URL content to the downstream proxy server and parses the URL content to obtain the embedded object prior to receiving a corresponding embedded object request message initiated by the web browser,

wherein the upstream proxy server comprises means for using a cookie when obtaining the embedded object.

* * * * *

US 20020091763A1

(19) **United States**

(12) **Patent Application Publication** (10) **Pub. No.: US 2002/0091763 A1**

Shah et al. (43) **Pub. Date: Jul. 11, 2002**

(54) **CLIENT-SIDE PERFORMANCE OPTIMIZATION SYSTEM FOR STREAMED APPLICATIONS**

(76) Inventors: **Lacky Vasant Shah**, Fremont, CA (US); **Daniel Takeo Arai**, Sunnyvale, CA (US); **Manuel Enrique Benitez**, Cupertino, CA (US); **Anne Marie Holler**, Santa Clara, CA (US); **Robert Curtis Wohlgemuth**, Santa Clara, CA (US)

Correspondence Address:
GLENN PATENT GROUP
3475 EDISON WAY
SUITE L
MENLO PARK, CA 94025 (US)

(21) Appl. No.: **09/858,260**

(22) Filed: **May 15, 2001**

Related U.S. Application Data

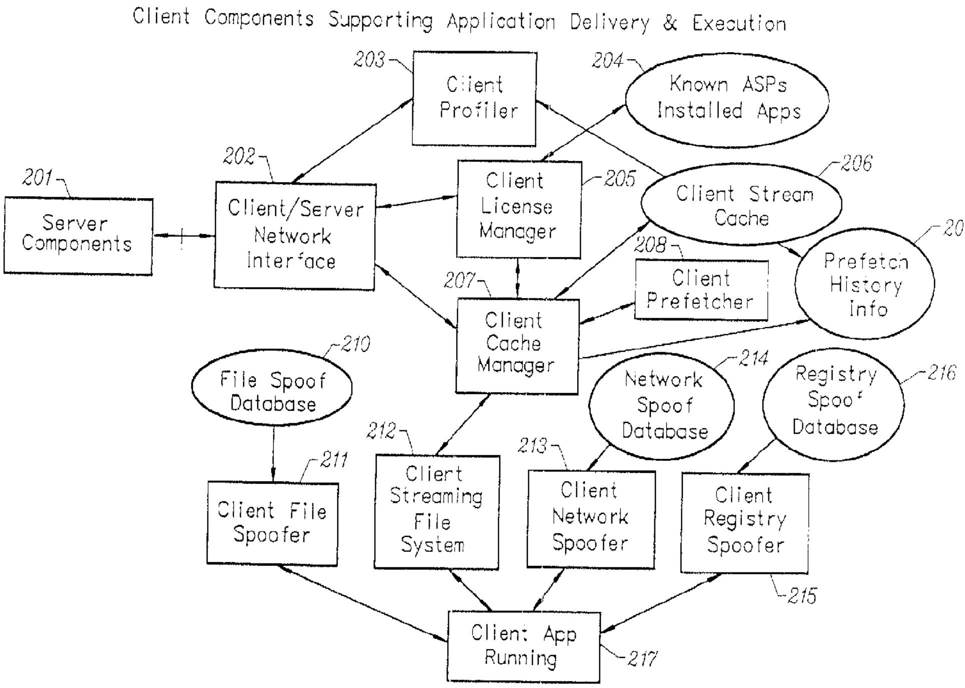
(63) Non-provisional of provisional application No. 60/246,384, filed on Nov. 6, 2000.

Publication Classification

(51)	**Int. Cl.^{7}**	**G06F 15/16**
(52)	**U.S. Cl.**	**709/203**

(57) **ABSTRACT**

An client-side performance optimization system for streamed applications provides several approaches for fulfilling client-side application code and data file requests for streamed applications. A streaming file system or file driver is installed on the client system that receives and fulfills application code and data requests from a persistent cache or the streaming application server. The client or the server can initiate the prefetching of application code and data to improve interactive application performance. A client-to-client communication mechanism allows local application customization to travel from one client machine to another without involving server communication. Applications are patched or upgraded via a change in the root directory for that application. The client can be notified of application upgrades by the server which can be marked as mandatory, in which case the client will force the application to be upgraded. The server broadcasts an application program's code and data and any client that is interested in that particular application program stores the broadcasted code and data for later use.



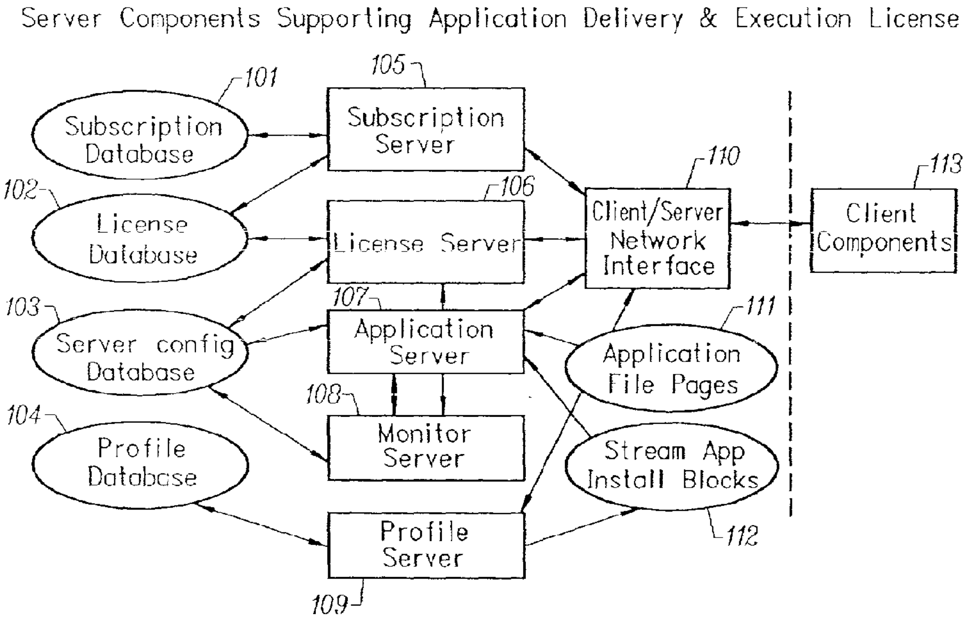

Server Components Supporting Application Delivery & Execution License
101
Subscription Database
105
Subscription Server
102
License Database
106
License Server
110
Client/Server Network Interface
113
Client Components
103
Server config Database
107
Application Server
111
Application File Pages
108
Monitor Server
104
Profile Database
Stream App Install Blocks
112
Profile Server
109

FIG. 1

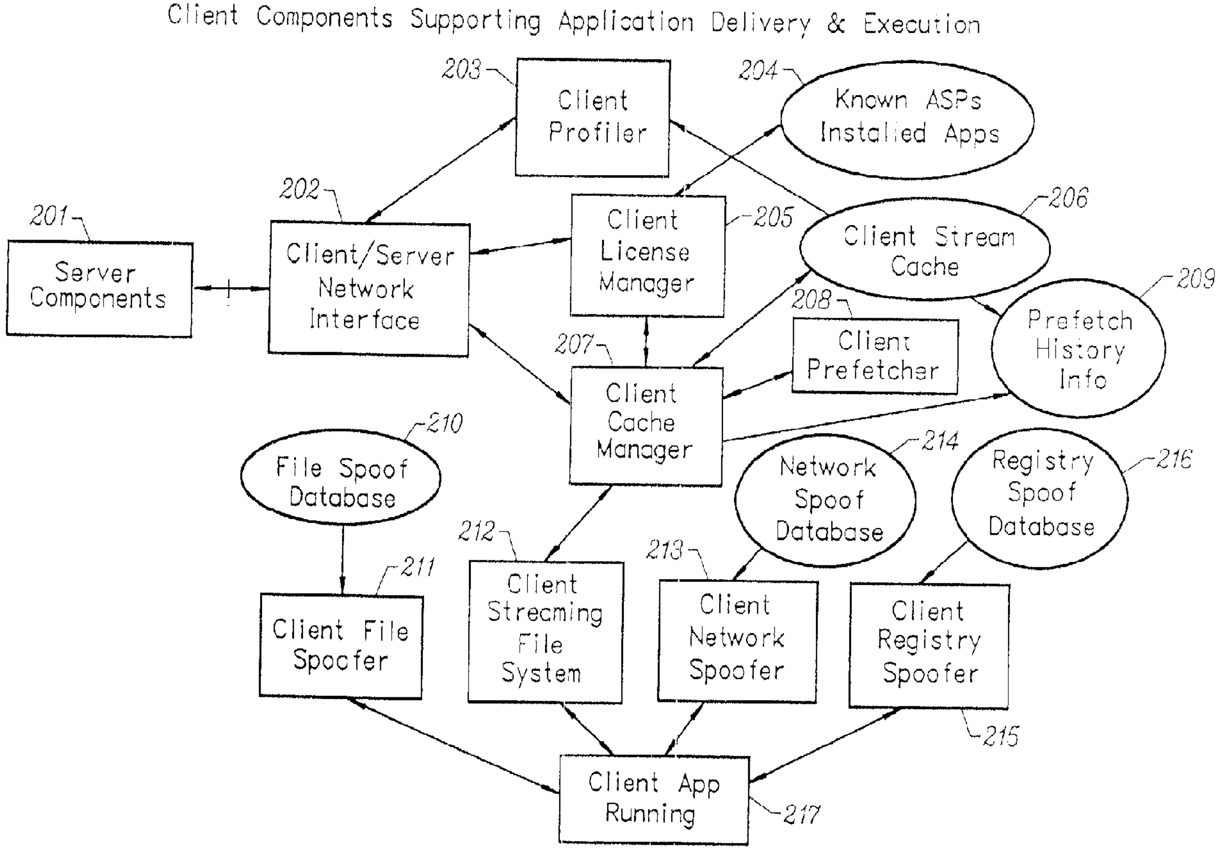

Client Components Supporting Application Delivery & Execution
201
Server Components
202
Client/Server Network Interface
203
Client Profiler
204
Known ASPs Installed Apps
205
Client License Manager
206
Client Stream Cache
207
Client Cache Manager
208
Client Prefetcher
209
Prefetch History Info
210
File Spoof Database
211
Client File Spoofer
212
Client Streaming File System
213
Client Network Spoofer
214
Network Spoof Database
215
Client Registry Spoofer
216
Registry Spoof Database
217
Client App Running

FIG. 2

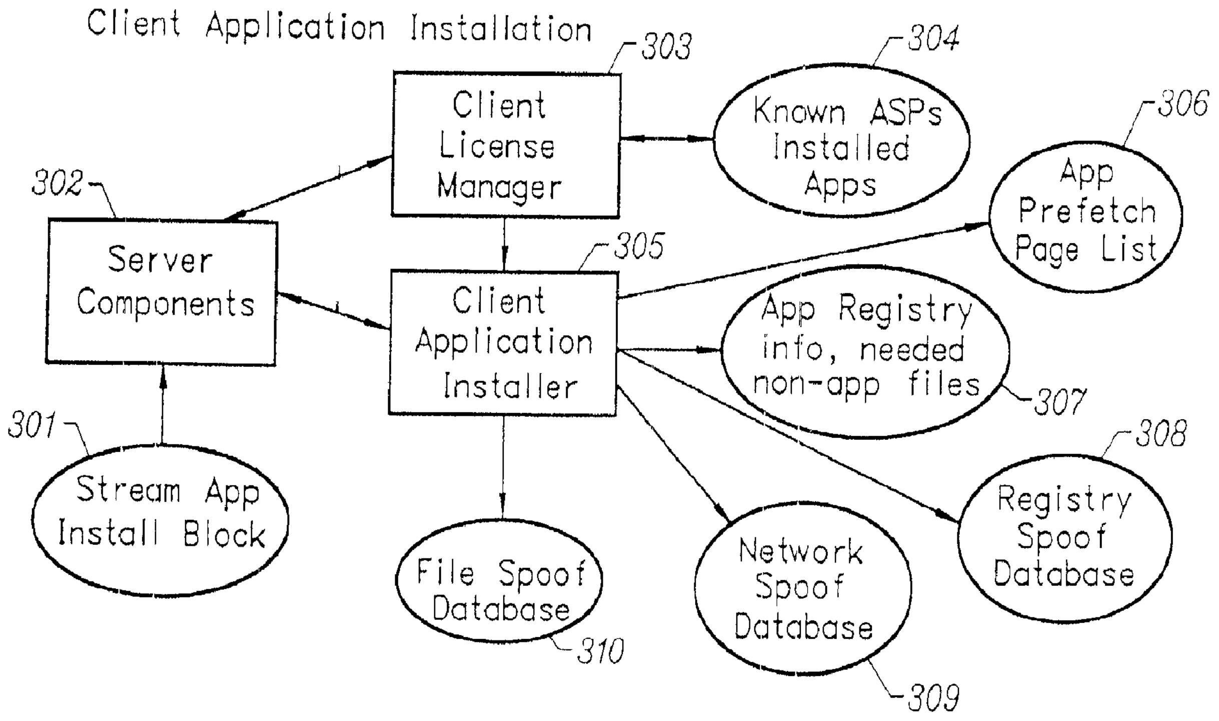

Client Application Installation
301
Stream App Install Block
302
Server Components
303
Client License Manager
304
Known ASPs Installed Apps
305
Client Application Installer
306
App Prefetch Page List
307
App Registry info, needed non-app files
308
Registry Spoof Database
309
Network Spoof Database
310
File Spoof Database

FIG. 3

Application Stream Builder Input/Output

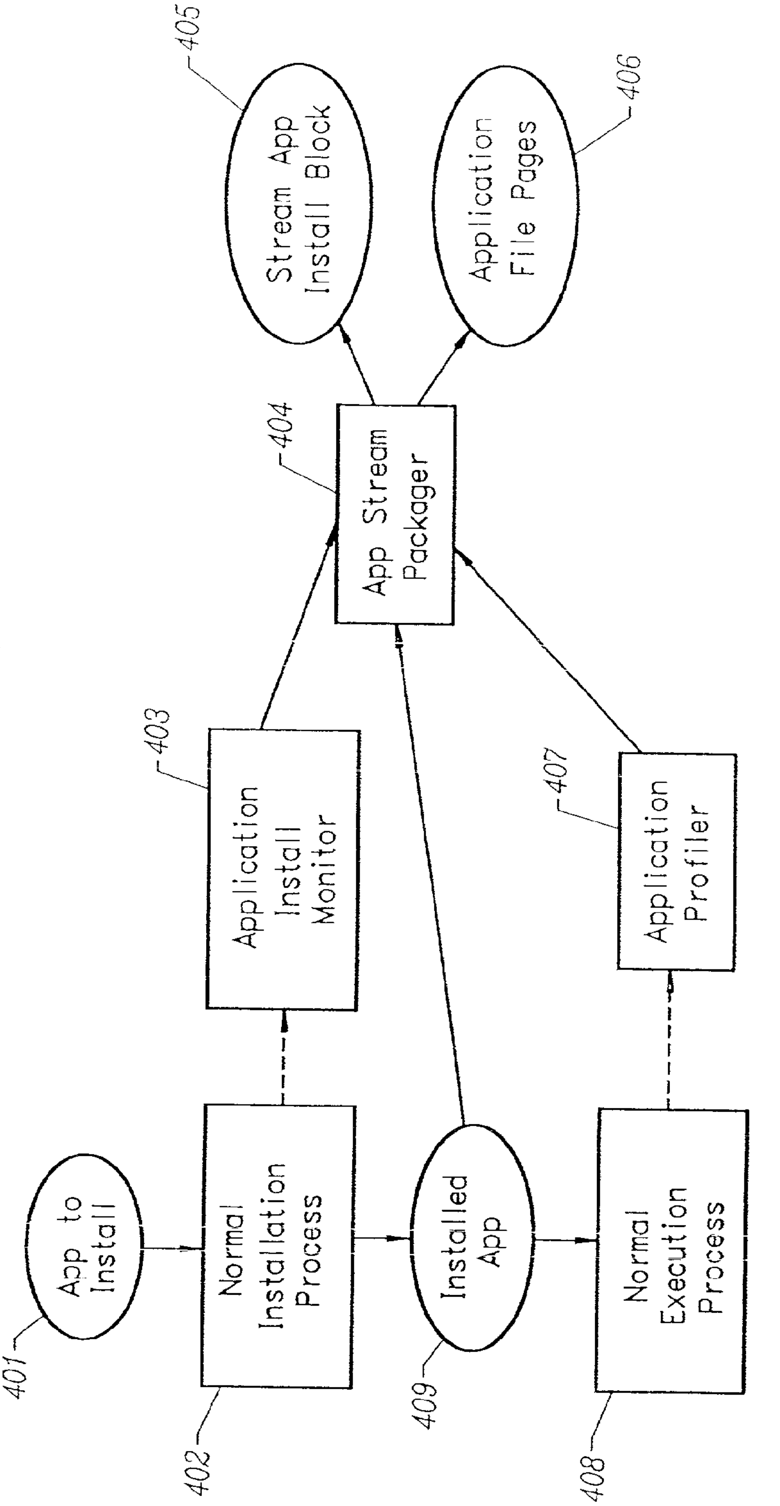


FIG. 4

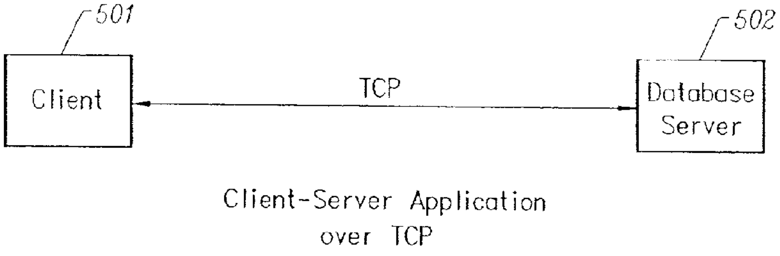

501
Client
TCP
502
Database Server
Client-Server Application over TCP

FIG. 5A

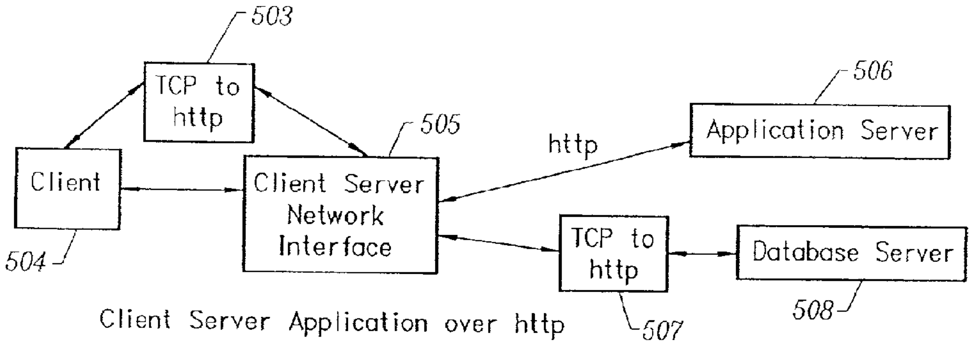

503
TCP to http
506
505
Application Server
http
Client
Client Server Network Interface
TCP to http
Database Server
504
508
507
Client Server Application over http

FIG. 5B

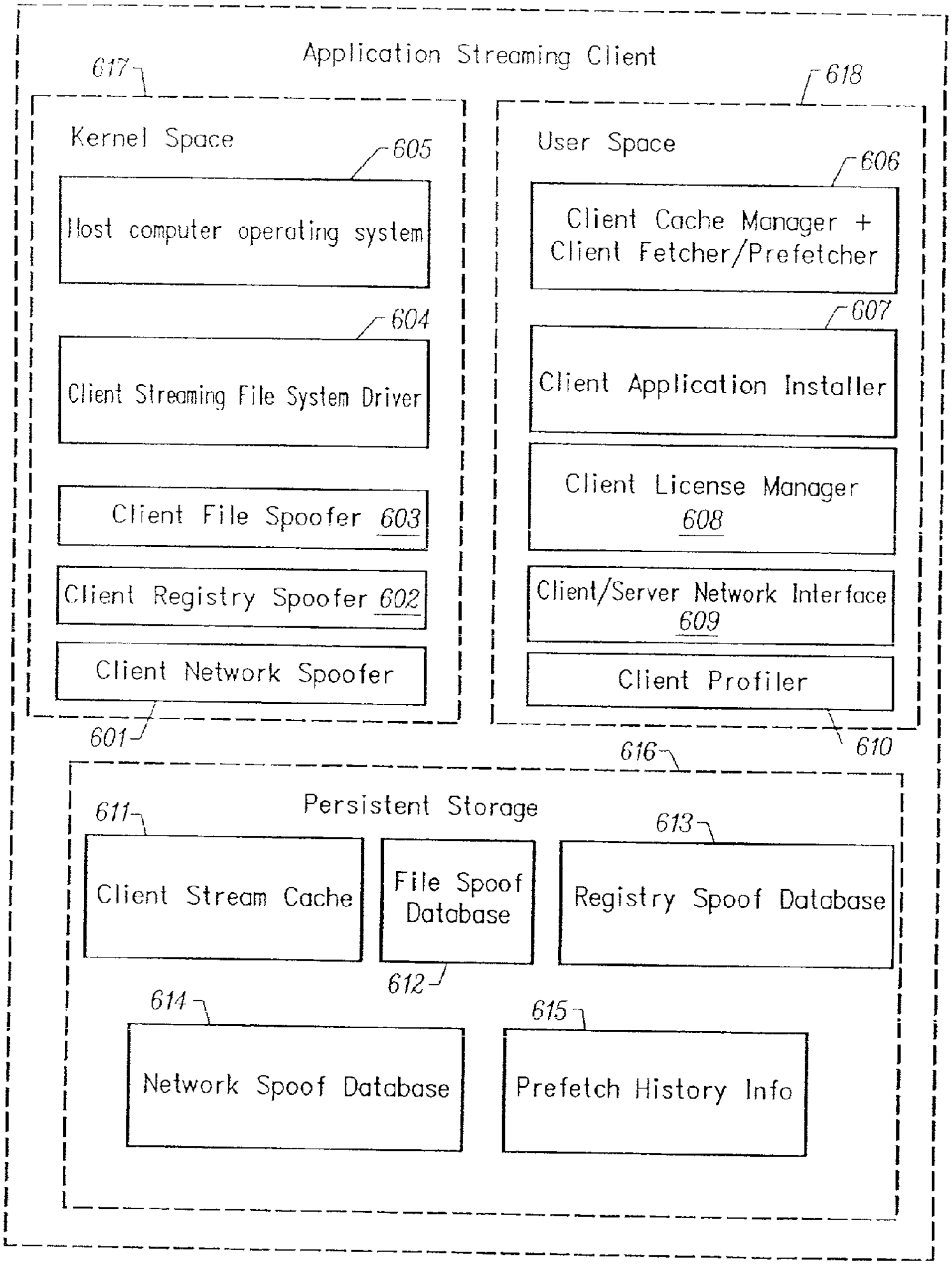

Application Streaming Client
617
Kernel Space
605
Host computer operating system
604
Client Streaming File System Driver
Client File Spoofer 603
Client Registry Spoofer 602
Client Network Spoofer
601
618
User Space
606
Client Cache Manager + Client Fetcher/Prefetcher
607
Client Application Installer
Client License Manager 608
Client/Server Network Interface 609
Client Profiler
610
616
Persistent Storage
611
Client Stream Cache
File Spoof Database
612
613
Registry Spoof Database
614
Network Spoof Database
615
Prefetch History Info

FIG. 6A

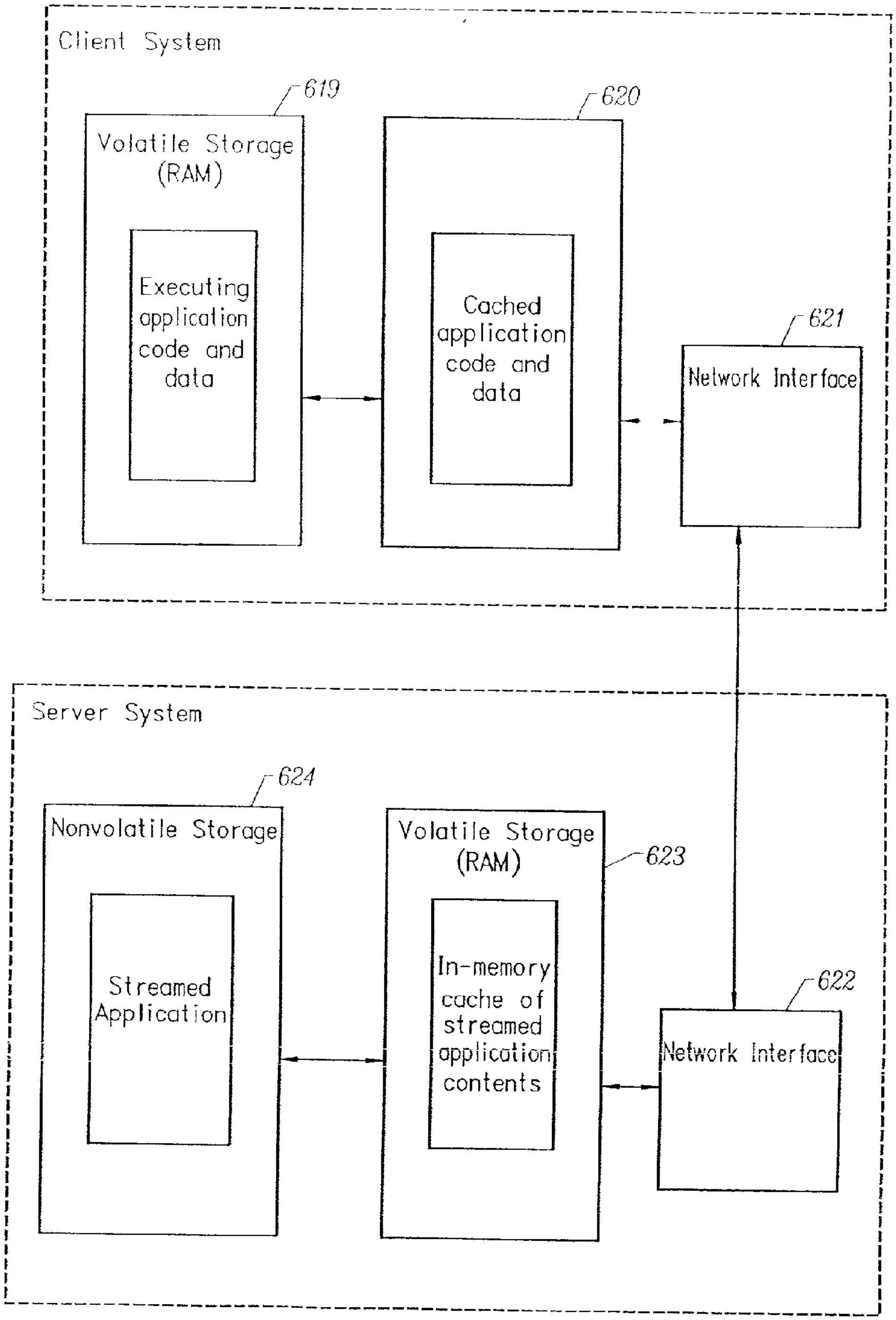

Client System
619
Volatile Storage (RAM)
Executing application code and data
620
Cached application code and data
621
Network Interface
Server System
624
Nonvolatile Storage
Streamed Application
Volatile Storage (RAM)
623
In-memory cache of streamed application contents
622
Network Interface

FIG. 6B

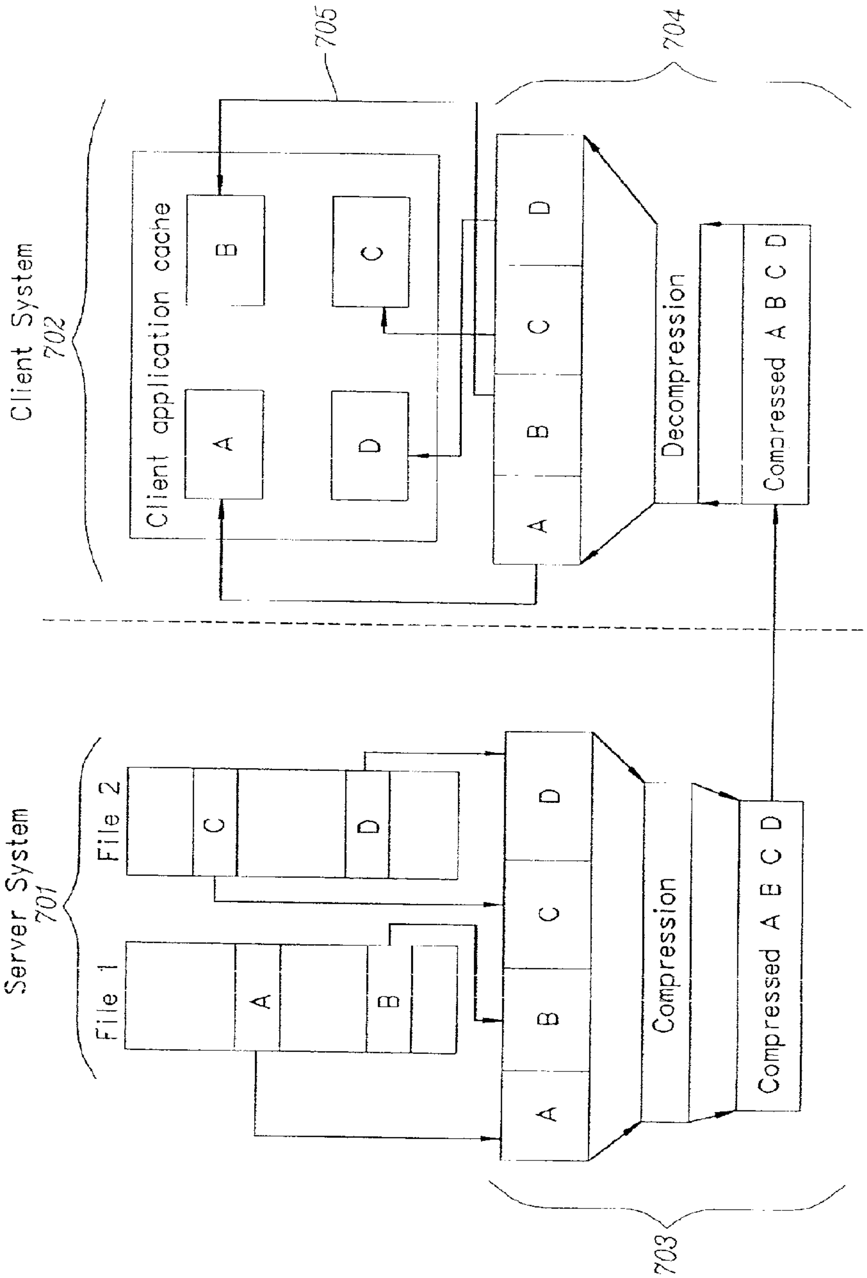

Server System
701
File 1
A
B
File 2
C
D
A
B
C
D
Compression
Compressed A B C D
703
Client System
702
Client application cache
A
B
D
C
705
A
B
C
D
Decompression
Compressed A B C D
704

FIG. 7A

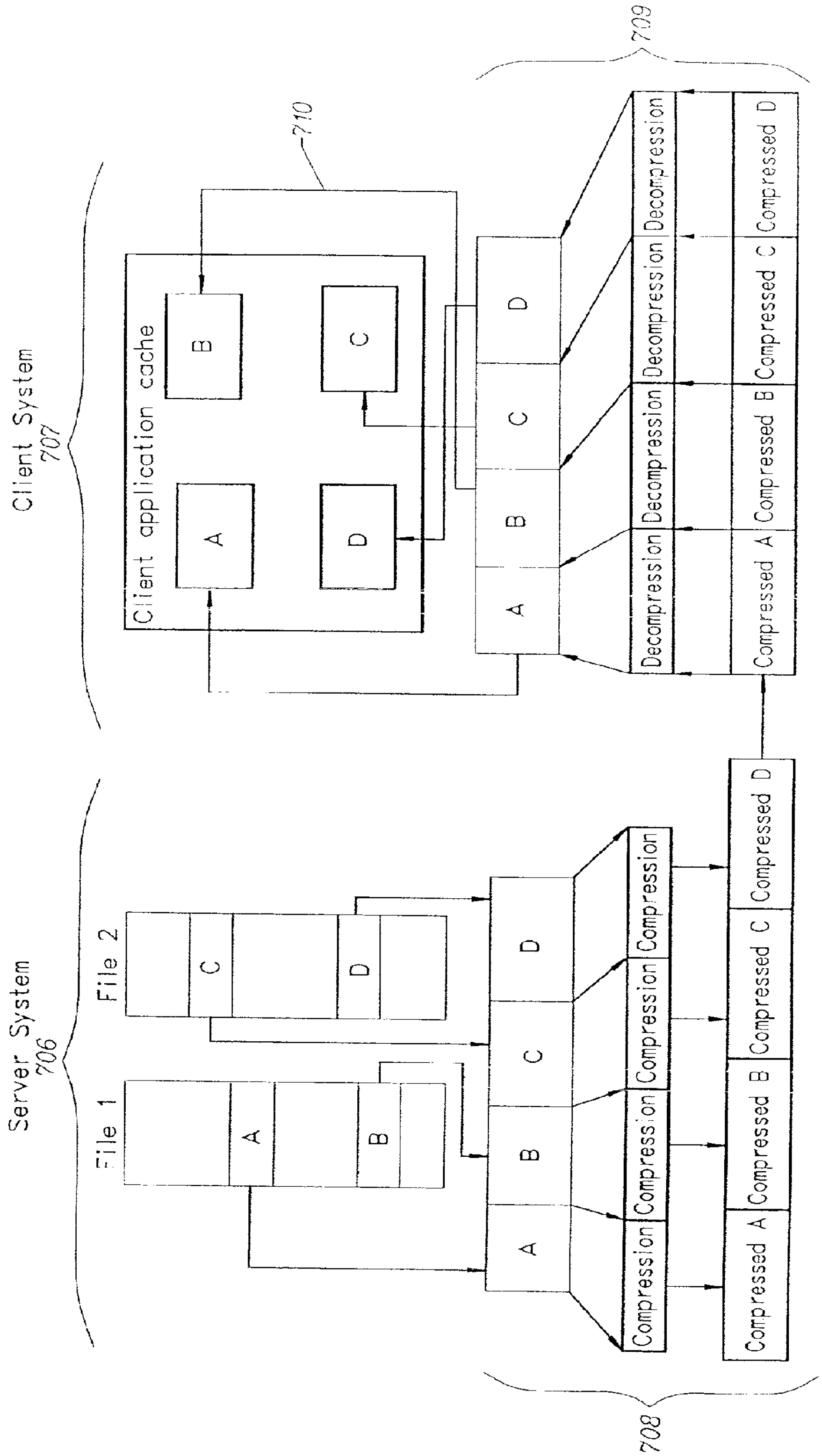

Server System
706
File 1
A
B
File 2
C
D
A
B
C
D
Compression
Compression
Compression
Compression
Compressed A
Compressed B
Compressed C
Compressed D
708
Client System
707
Client application cache
A
B
D
C
710
A
B
C
D
Decompression
Decompression
Decompression
Decompression
Compressed A
Compressed B
Compressed C
Compressed D
709

FIG. 7B

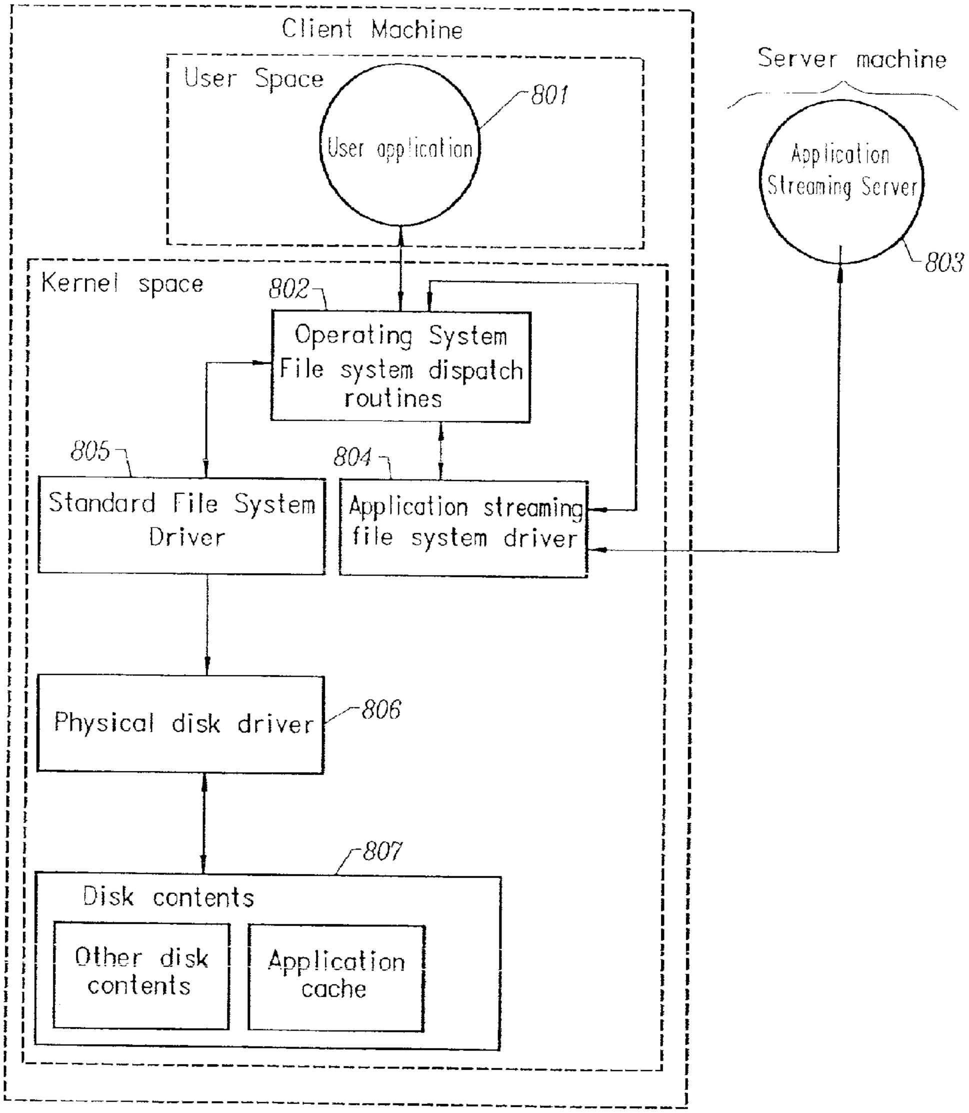

Client Machine
User Space
User application
801
Server machine
Application Streaming Server
803
Kernel space
802
Operating System File system dispatch routines
805
Standard File System Driver
804
Application streaming file system driver
Physical disk driver
806
807
Disk contents
Other disk contents
Application cache

FIG. 8

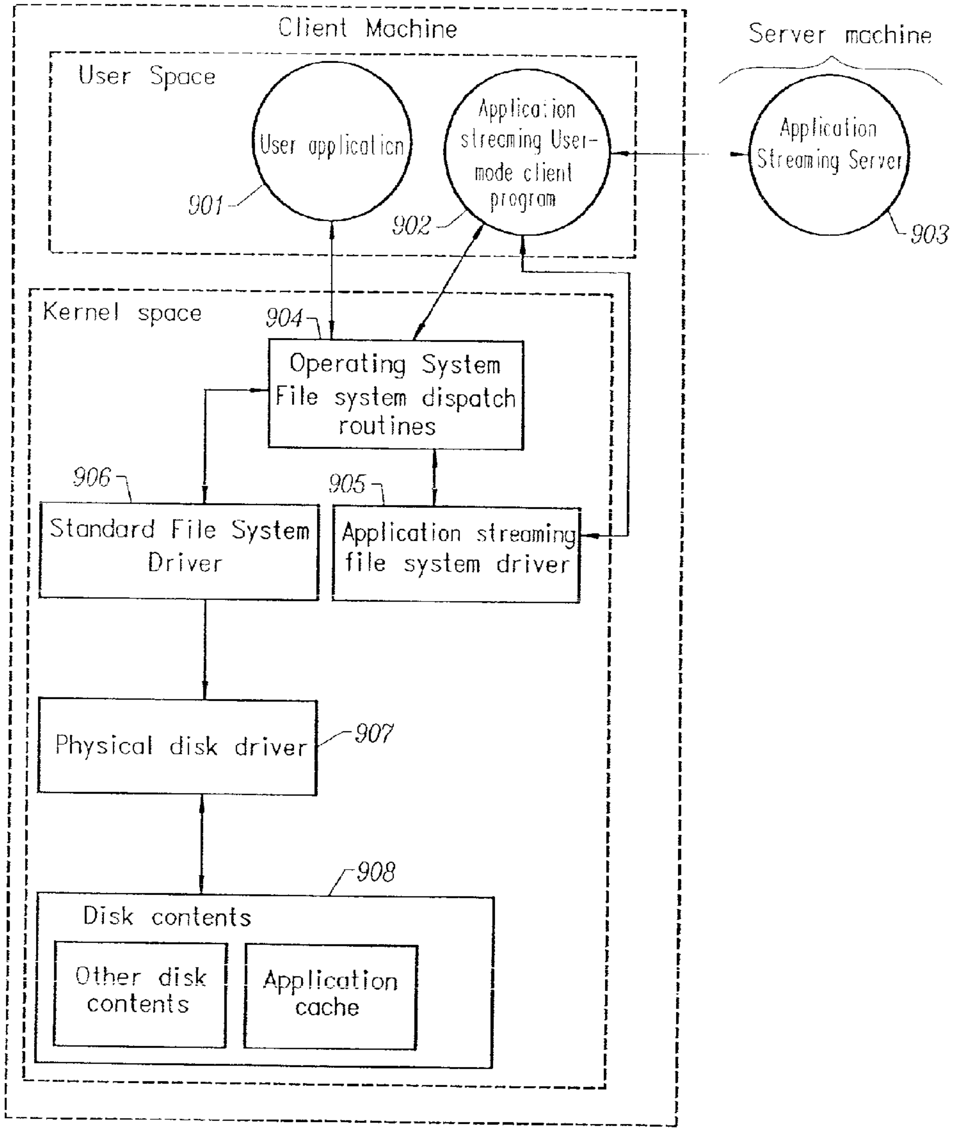

Client Machine
User Space
User application
901
Application streaming User-mode client program
902
Server machine
Application Streaming Server
903
Kernel space
904
Operating System File system dispatch routines
906
Standard File System Driver
905
Application streaming file system driver
Physical disk driver
907
908
Disk contents
Other disk contents
Application cache

FIG. 9

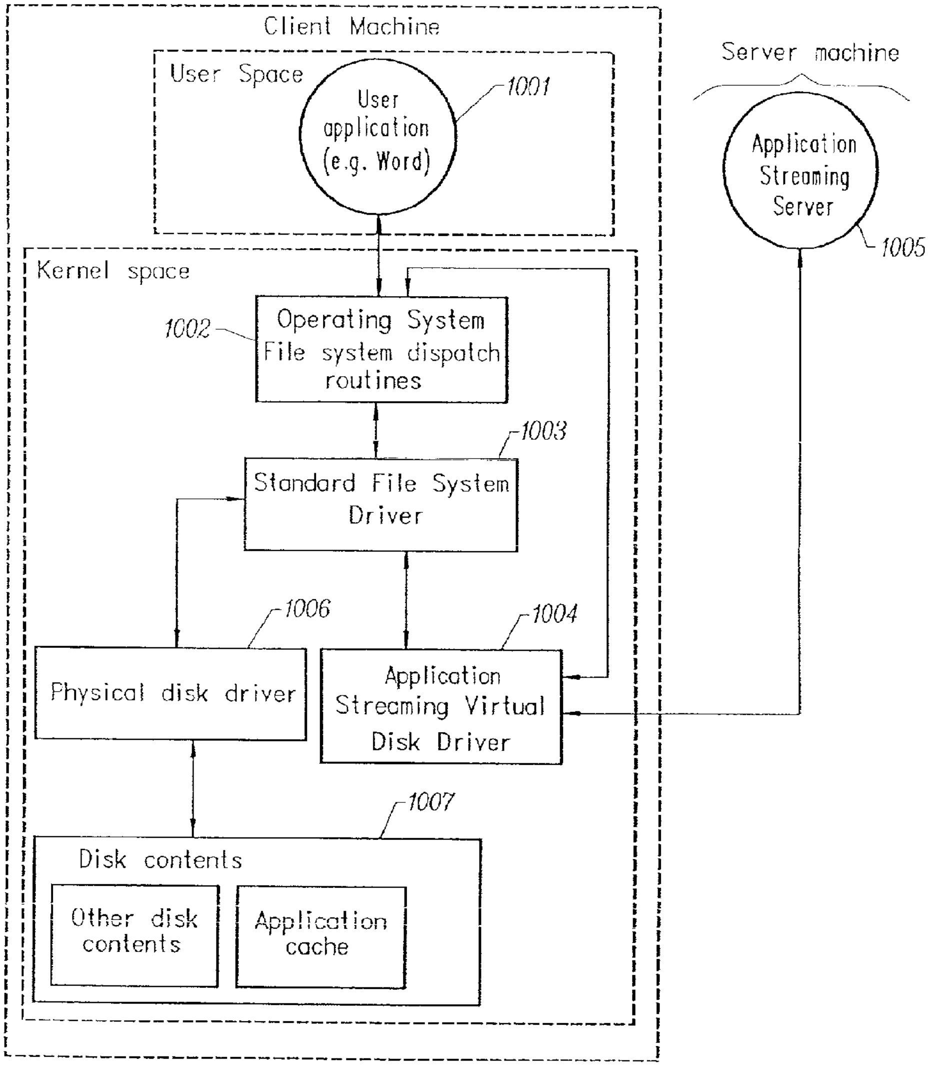

Client Machine
User Space
User application (e.g. Word)
1001
Server machine
Application Streaming Server
1005
Kernel space
1002
Operating System File system dispatch routines
1003
Standard File System Driver
1006
1004
Physical disk driver
Application Streaming Virtual Disk Driver
1007
Disk contents
Other disk contents
Application cache

FIG. 10

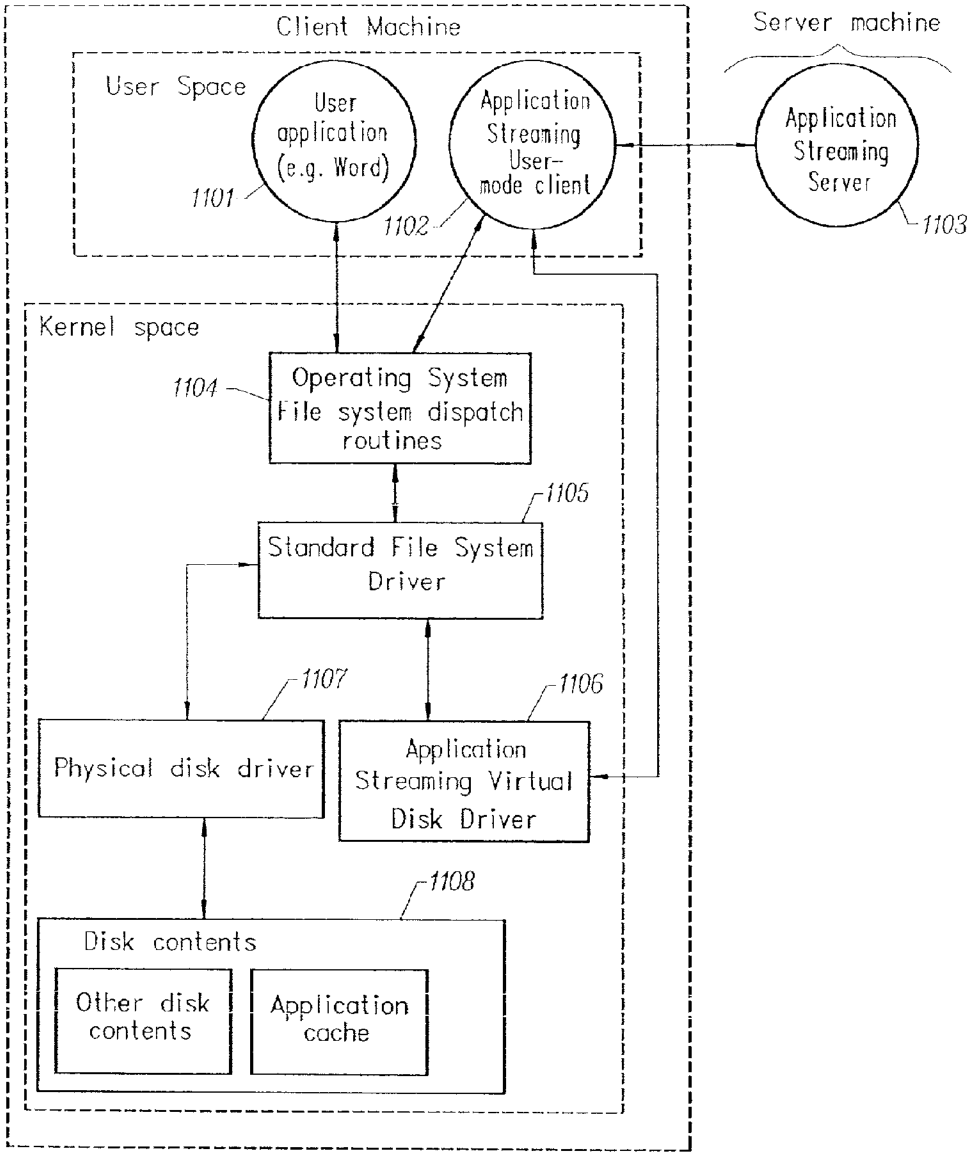

Client Machine
User Space
User application (e.g. Word)
1101
Application Streaming User-mode client
1102
Server machine
Application Streaming Server
1103
Kernel space
1104
Operating System File system dispatch routines
1105
Standard File System Driver
1107
1106
Physical disk driver
Application Streaming Virtual Disk Driver
1108
Disk contents
Other disk contents
Application cache

FIG. 11

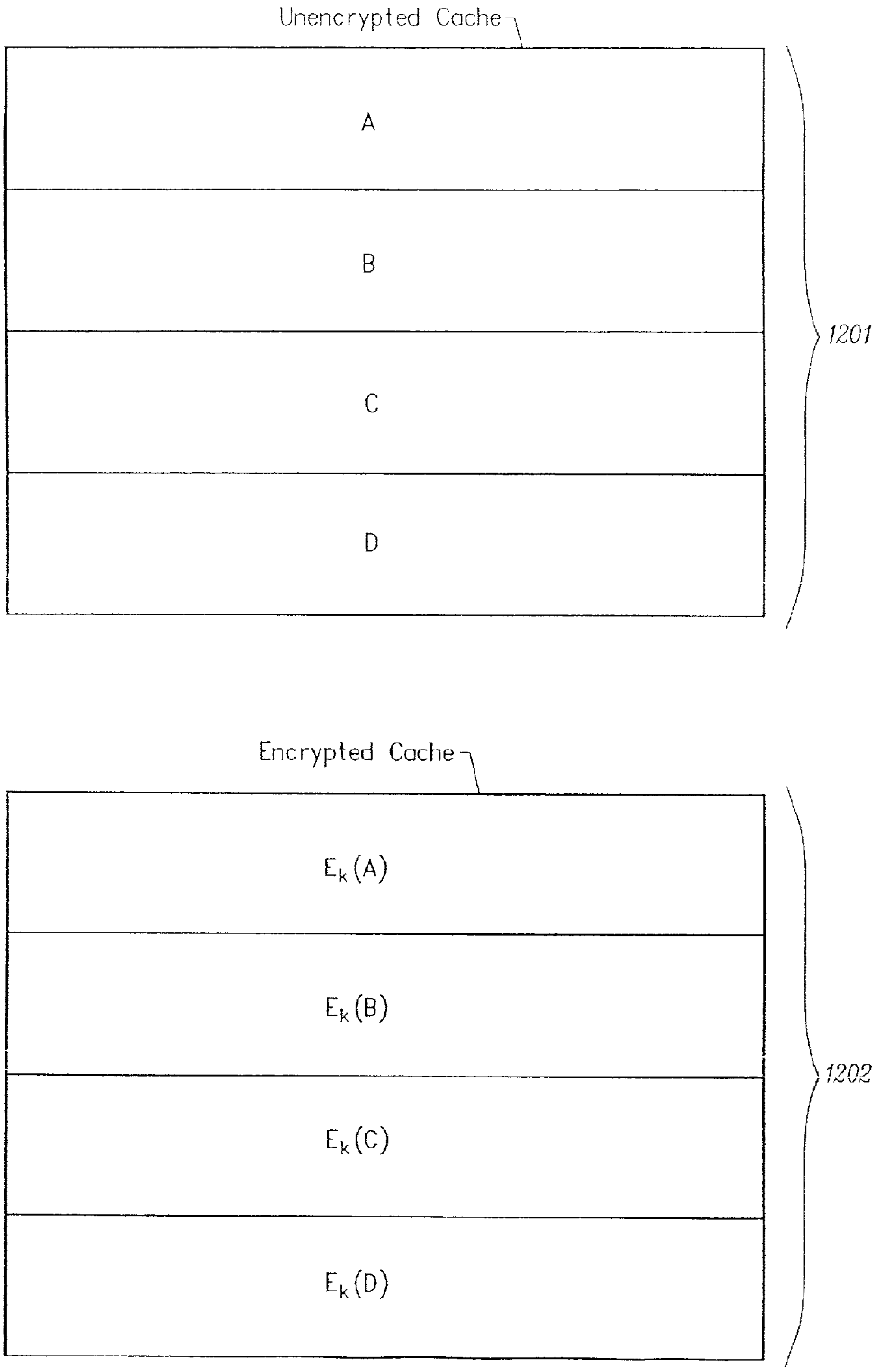

Unencrypted Cache
A
B
C
D
1201
Encrypted Cache
$E_k(A)$
$E_k(B)$
$E_k(C)$
$E_k(D)$
1202

FIG. 12

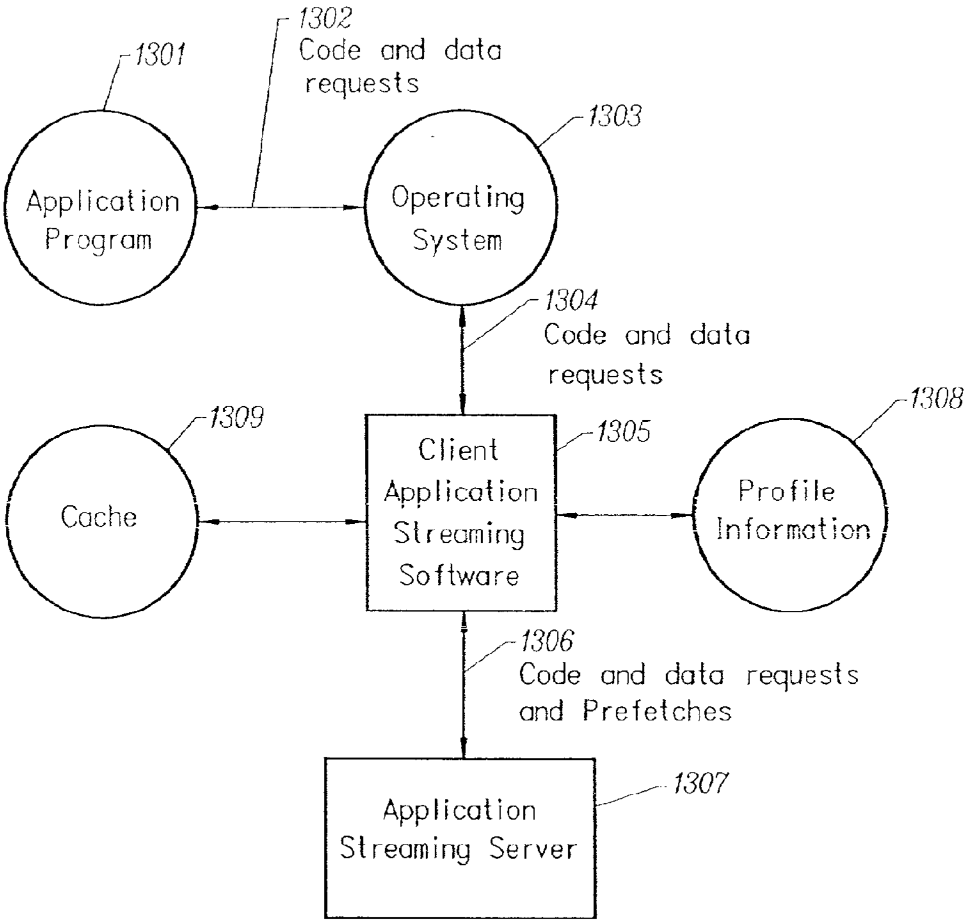

1301
Application Program
1302
Code and data requests
1303
Operating System
1304
Code and data requests
1309
Cache
1305
Client Application Streaming Software
1308
Profile Information
1306
Code and data requests and Prefetches
1307
Application Streaming Server

FIG. 13

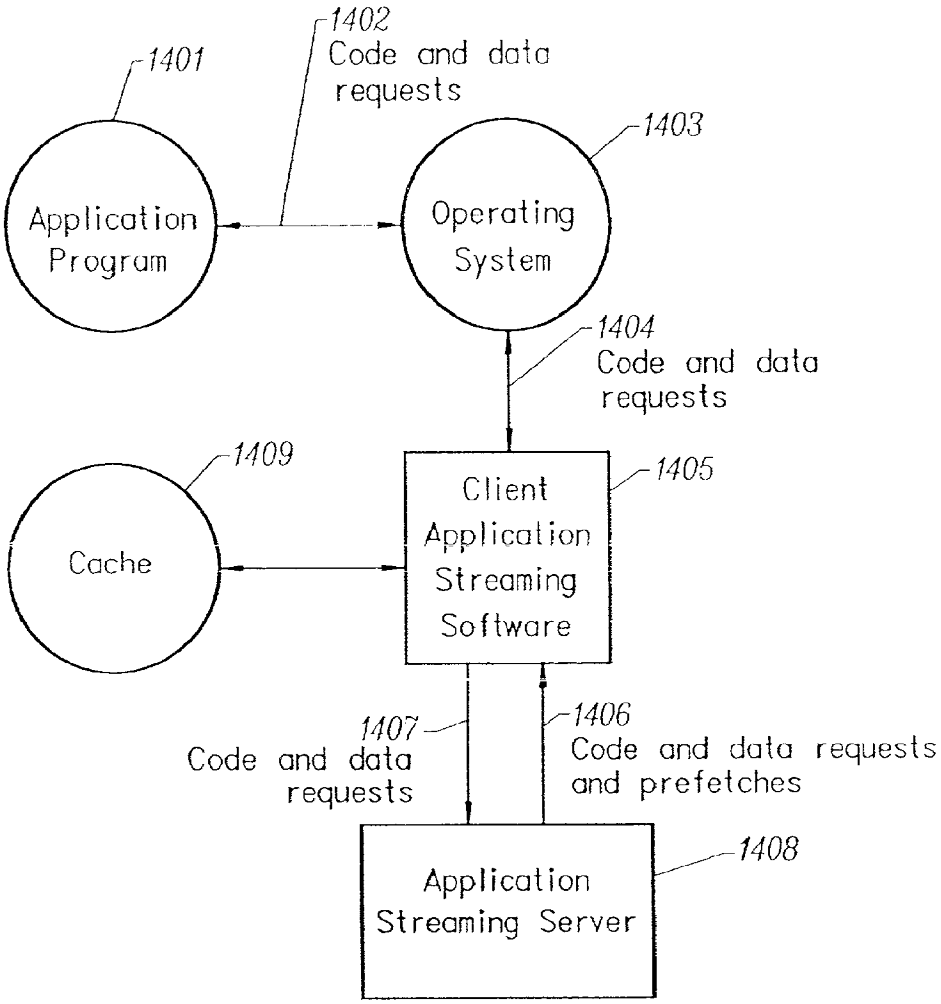

1401
Application Program
1402
Code and data requests
1403
Operating System
1404
Code and data requests
1409
Cache
1405
Client Application Streaming Software
1407
Code and data requests
1406
Code and data requests and prefetches
1408
Application Streaming Server

FIG. 14

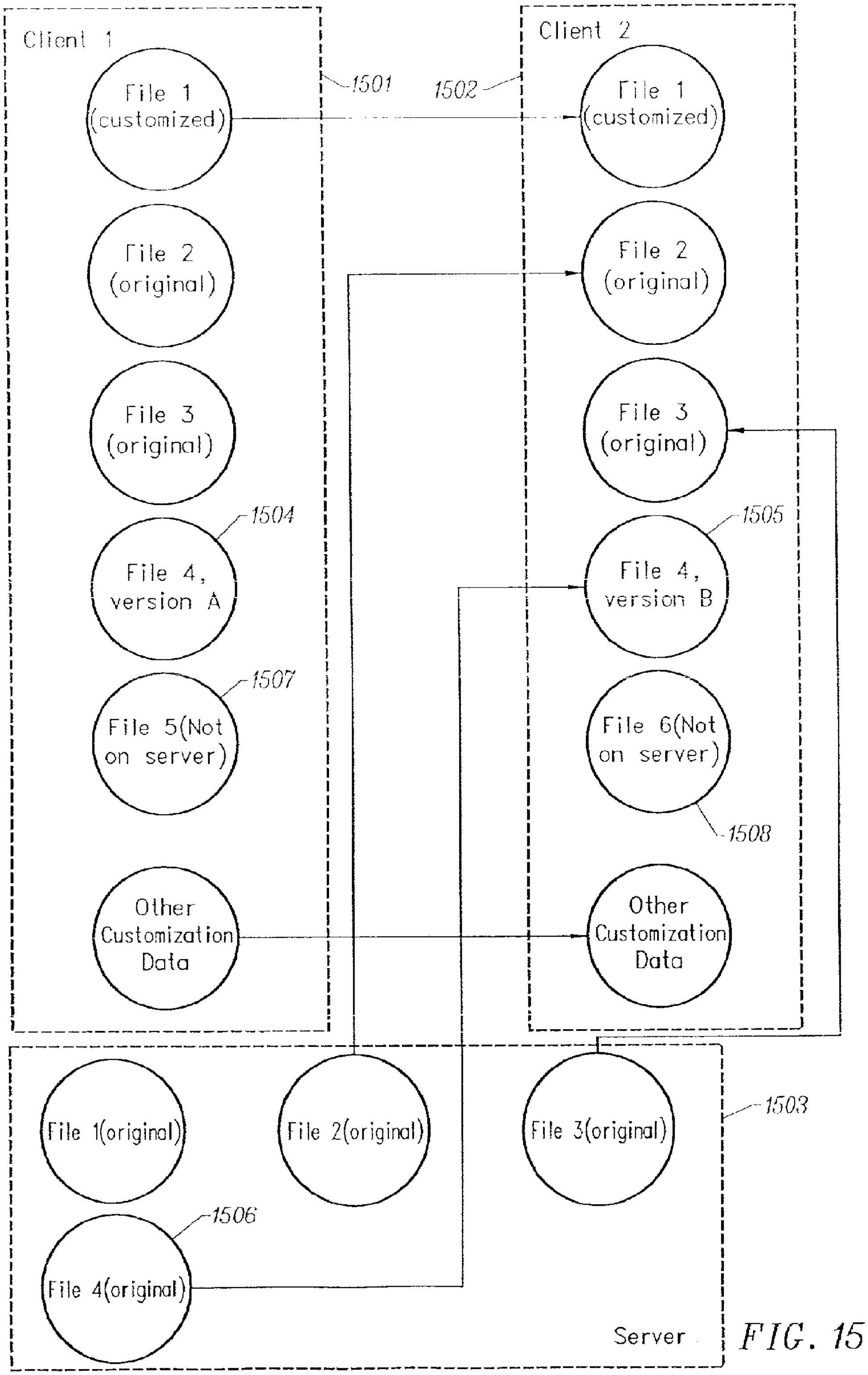

Client 1
File 1 (customized)
File 2 (original)
File 3 (original)
File 4, version A
File 5(Not on server)
Other Customization Data
1501
1502
1504
1507
Client 2
File 1 (customized)
File 2 (original)
File 3 (original)
File 4, version B
File 6(Not on server)
Other Customization Data
1505
1508
File 1(original)
File 2(original)
File 3(original)
File 4(original)
1506
1503
Server

FIG. 15

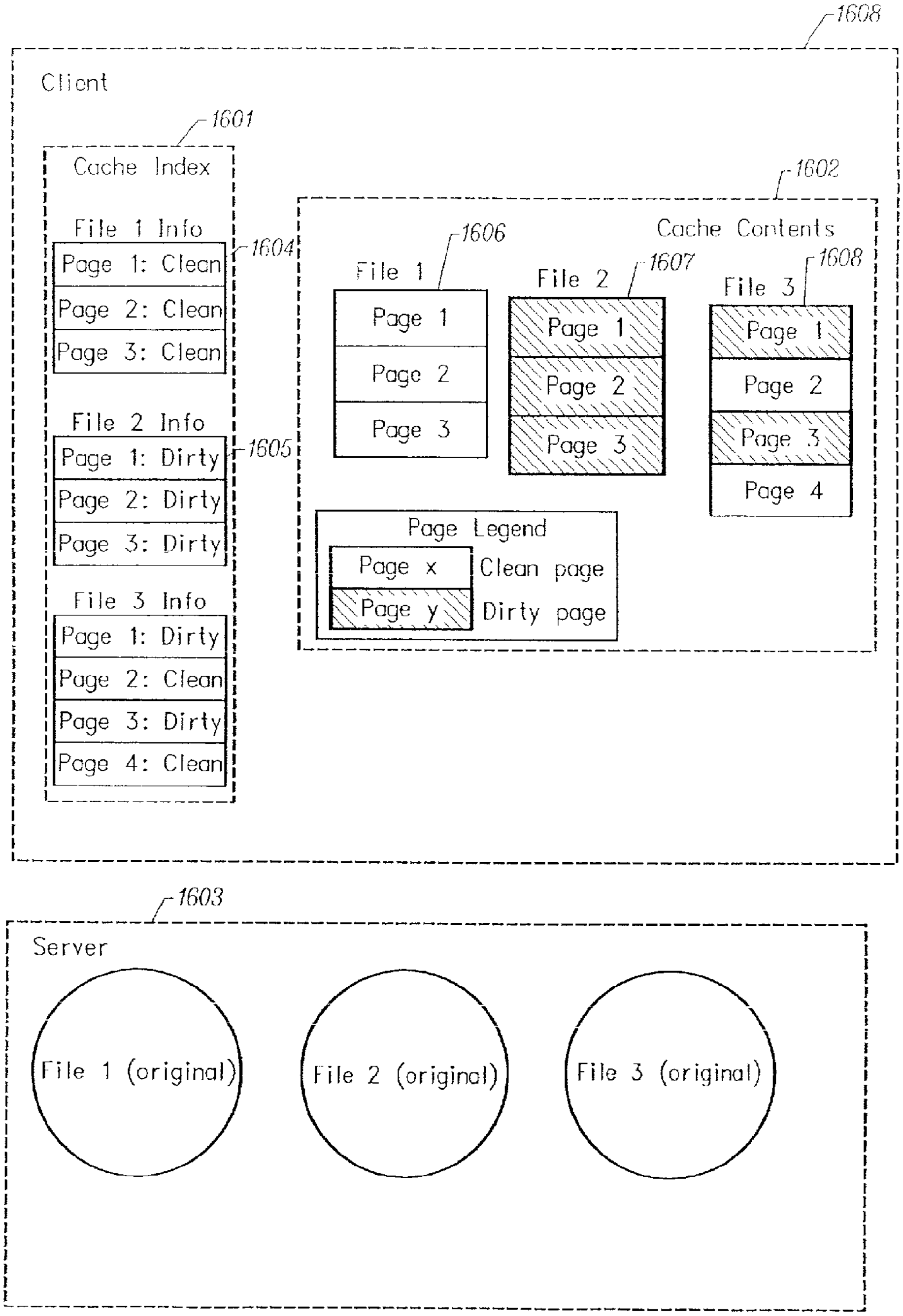
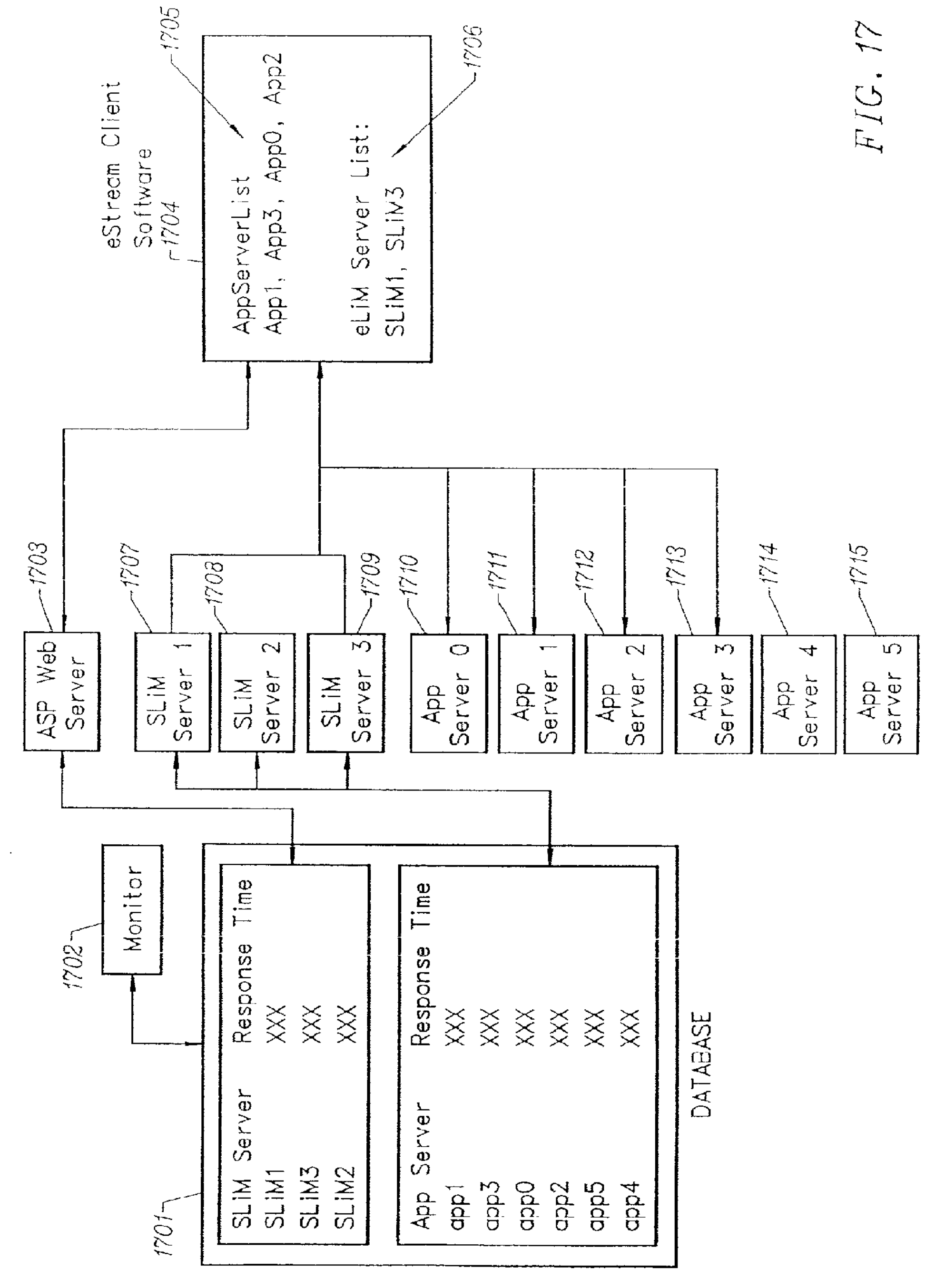

1608
Client
1601
Cache Index
File 1 Info
Page 1: Clean
Page 2: Clean
Page 3: Clean
1604
File 2 Info
Page 1: Dirty
Page 2: Dirty
Page 3: Dirty
1605
File 3 Info
Page 1: Dirty
Page 2: Clean
Page 3: Dirty
Page 4: Clean
1602
Cache Contents
1606
File 1
Page 1
Page 2
Page 3
File 2
1607
Page 1
Page 2
Page 3
File 3
1608
Page 1
Page 2
Page 3
Page 4
Page Legend
Page x
Clean page
Page y
Dirty page
1603
Server
File 1 (original)
File 2 (original)
File 3 (original)

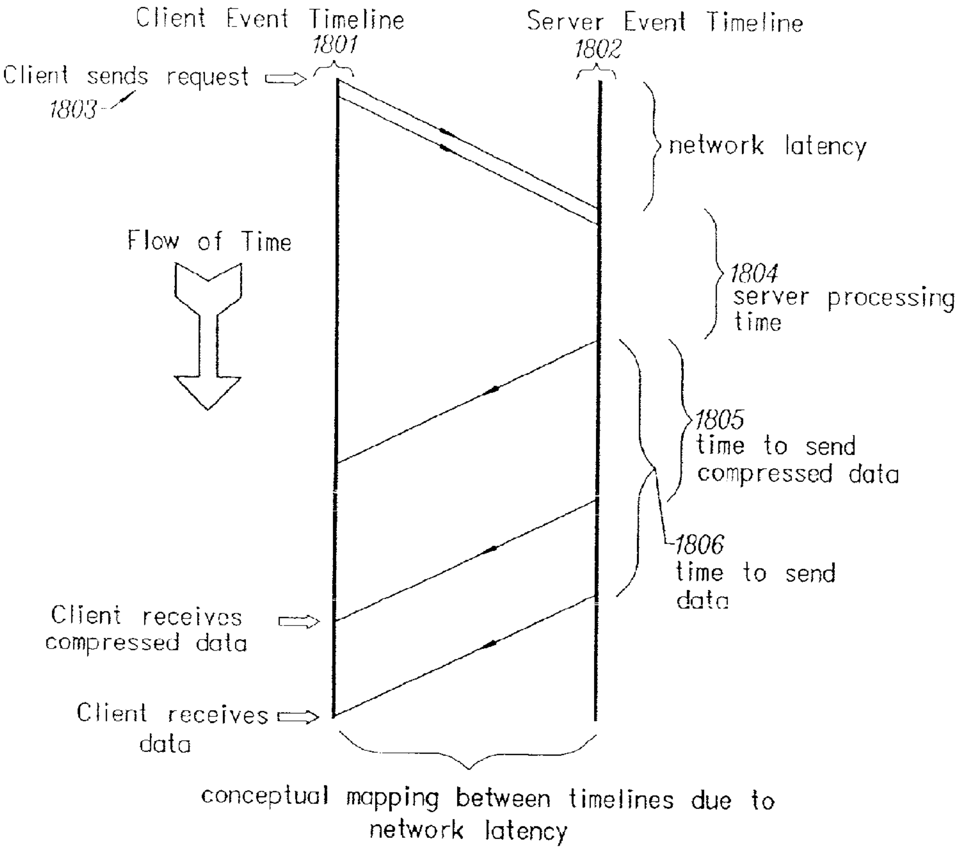
FIG. 16



SLiM Server	Response Time
SLiM1	XXX
SLiM3	XXX
SLiM2	XXX

App Server	Response Time
app1	XXX
app3	XXX
app0	XXX
app2	XXX
app5	XXX
app4	XXX

FIG. 17



Client receives data sooner if it is compressed

FIG. 18

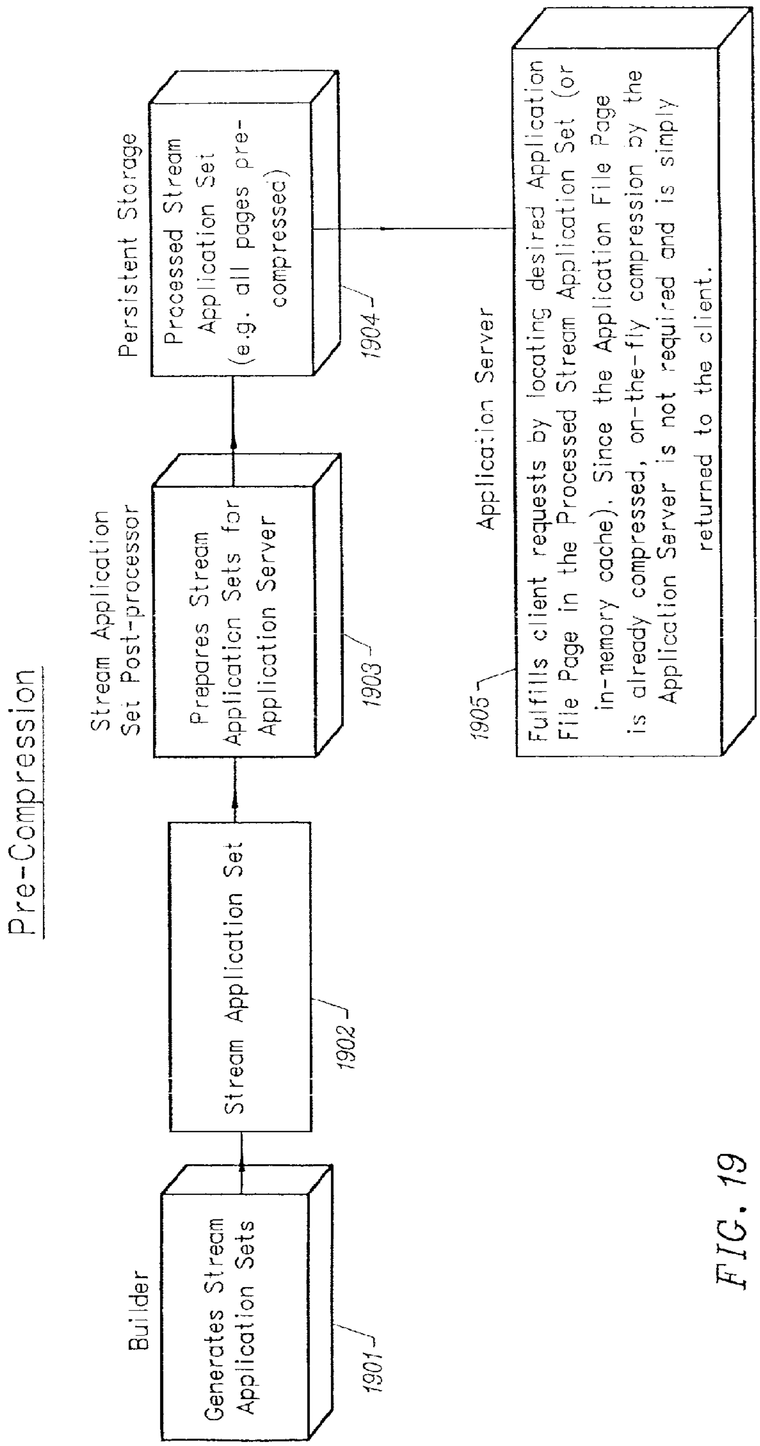
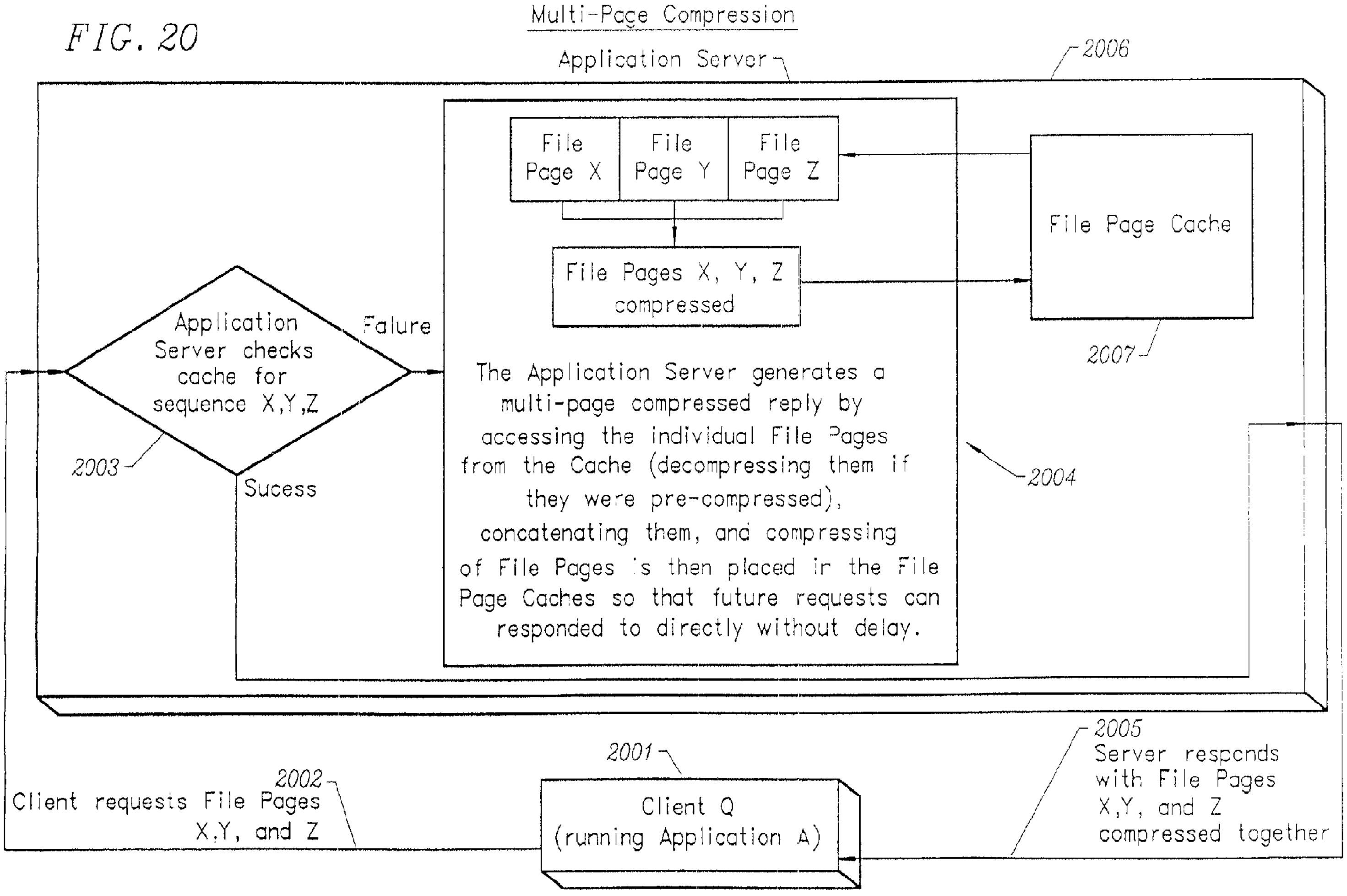

Pre-Compression
Builder
Generates Stream Application Sets
1901
Stream Application Set
1902
Stream Application Set Post-processor
Prepares Stream Application Sets for Application Server
1903
Persistent Storage
Processed Stream Application Set (e.g. all pages pre-compressed)
1904
Application Server
Fulfills client requests by locating desired Application File Page in the Processed Stream Application Set (or in-memory cache). Since the Application File Page is already compressed, on-the-fly compression by the Application Server is not required and is simply returned to the client.
1905

FIG. 19

FIG. 20


Multi-Page Compression
Application Server
2006
File Page X
File Page Y
File Page Z
File Page Cache
File Pages X, Y, Z compressed
2007
Application Server checks cache for sequence X,Y,Z
Falure
Sucess
2003
The Application Server generates a multi-page compressed reply by accessing the individual File Pages from the Cache (decompressing them if they were pre-compressed), concatenating them, and compressing of File Pages 's then placed in the File Page Caches so that future requests can responded to directly without delay.
2004
2001
Client Q (running Application A)
2002
Client requests File Pages X,Y, and Z
2005
Server responds with File Pages X,Y, and Z compressed together

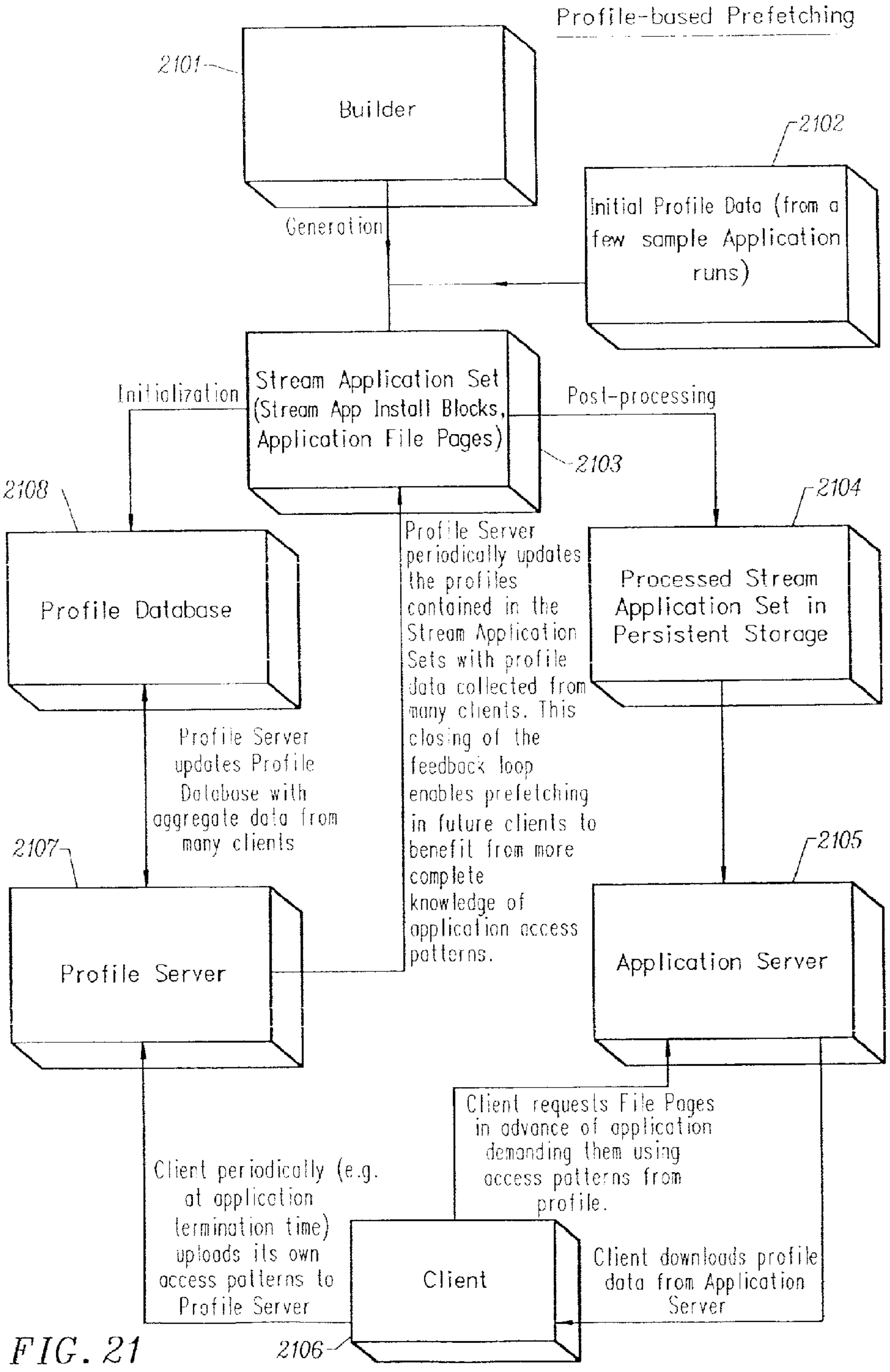

Profile-based Prefetching
2101
Builder
2102
Initial Profile Data (from a few sample Application runs)
Generation
Initialization
Stream Application Set (Stream App Install Blocks, Application File Pages)
2103
Post-processing
2108
Profile Database
2104
Processed Stream Application Set in Persistent Storage
Profile Server periodically updates the profiles contained in the Stream Application Sets with profile data collected from many clients. This closing of the feedback loop enables prefetching in future clients to benefit from more complete knowledge of application access patterns.
Profile Server updates Profile Database with aggregate data from many clients
2107
Profile Server
2105
Application Server
Client requests File Pages in advance of application demanding them using access patterns from profile.
Client periodically (e.g. at application termination time) uploads its own access patterns to Profile Server
Client
Client downloads profile data from Application Server
2106

FIG. 21

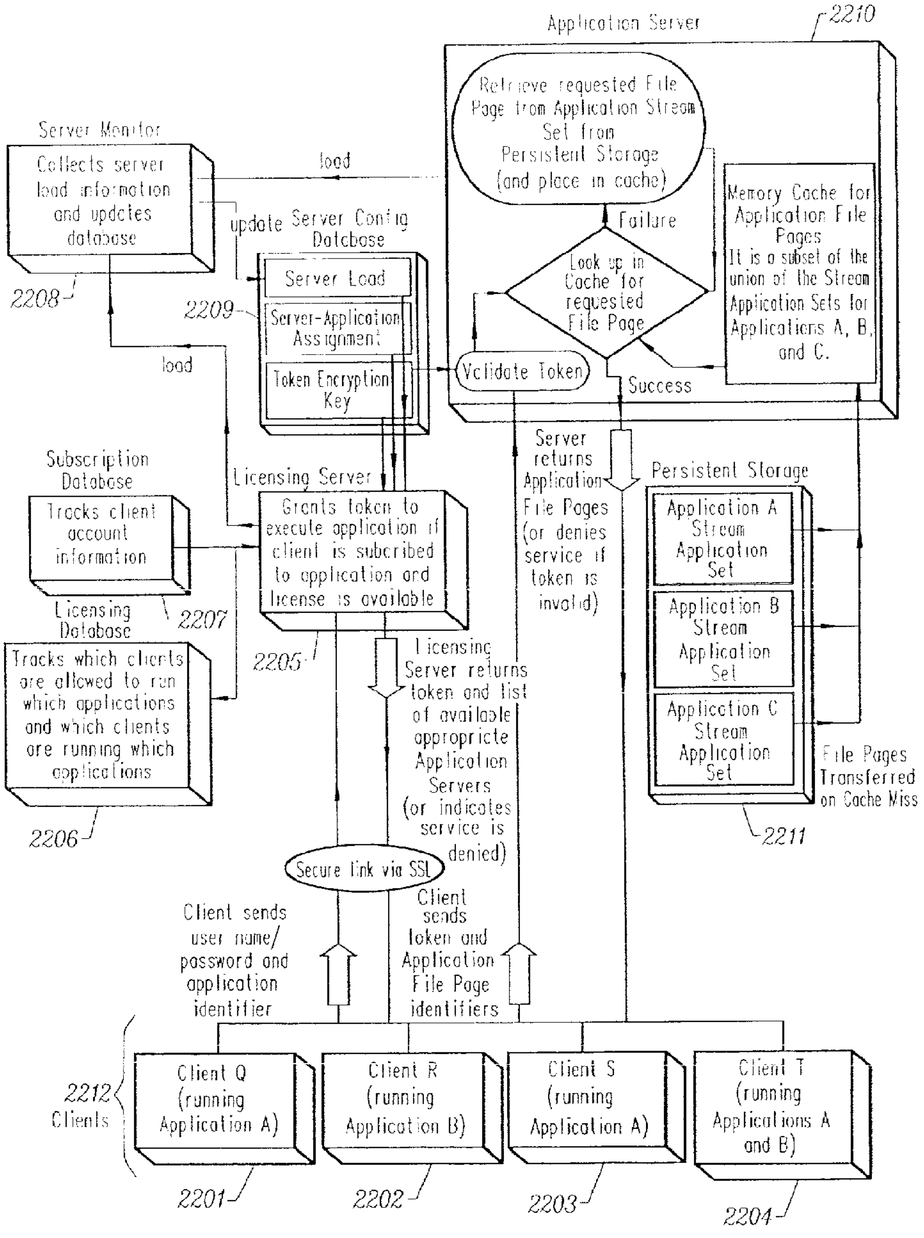

Application Server
2210
Retrieve requested File Page from Application Stream Set from Persistent Storage (and place in cache)
Failure
Look up in Cache for requested File Page
Memory Cache for Application File Pages It is a subset of the union of the Stream Application Sets for Applications A, B, and C.
Validate Token
Success
Server Monitor
Collects server load information and updates database
load
update
Server Config Database
Server Load
Server-Application Assignment
Token Encryption Key
2208
2209
load
Subscription Database
Tracks client account information
2207
Licensing Server
Grants token to execute application if client is subcribed to application and license is available
2205
Licensing Database
Tracks which clients are allowed to run which applications and which clients are running which applications
2206
Licensing Server returns token and list of available appropricte Application Servers (or indicates service is denied)
Server returns Application File Pages (or denies service if token is invalid)
Persistent Storage
Application A Stream Application Set
Application B Stream Application Set
Application C Stream Application Set
File Pages Transferred on Cache Miss
2211
Secure link via SSL
Client sends user name/ password and application identifier
Client sends token and Application File Page identifiers
2212
Clients
Client Q (running Application A)
Client R (running Application B)
Client S (running Application A)
Client T (running Applications A and B)
2201
2202
2203
2204

FIG. 22

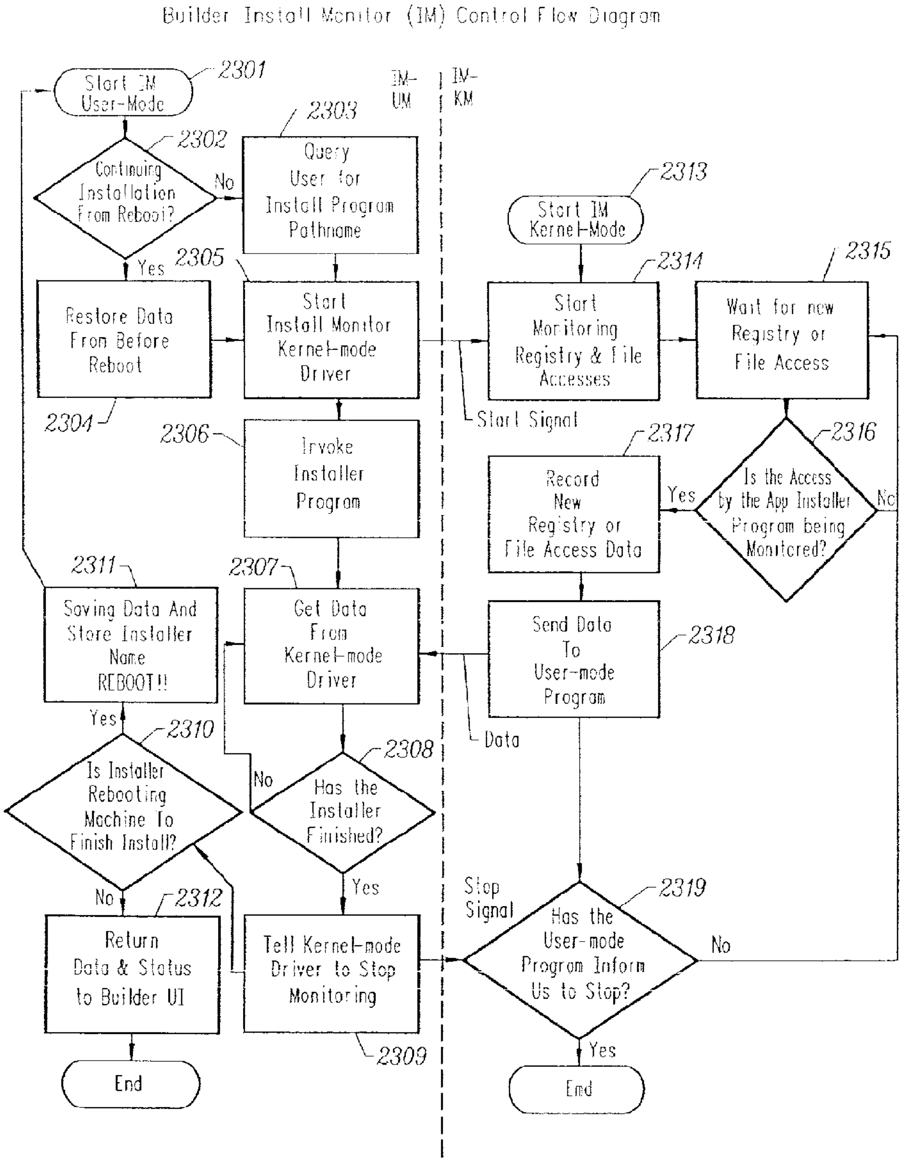

Builder Install Monitor (IM) Control Flow Diagram
IM-UM
IM-KM
Start IM User-Mode
2301
Continuing Installation From Reboot?
2302
No
Yes
Query User for Install Program Pathname
2303
Restore Data From Before Reboot
2304
Start Install Monitor Kernel-mode Driver
2305
Invoke Installer Program
2306
Get Data From Kernel-mode Driver
2307
Has the Installer Finished?
2308
No
Yes
Tell Kernel-mode Driver to Stop Monitoring
2309
Is Installer Rebooting Machine To Finish Install?
2310
Yes
No
Saving Data And Store Installer Name REBOOT!!
2311
Return Data & Status to Builder UI
2312
End
Start IM Kernel-Mode
2313
Start Monitoring Registry & File Accesses
2314
Start Signal
Wait for new Registry or File Access
2315
Is the Access by the App Installer Program being Monitored?
2316
Yes
No
Record New Registry or File Access Data
2317
Send Data To User-mode Program
2318
Data
Has the User-mode Program Inform Us to Stop?
2319
Stop Signal
No
Yes
End

FIG. 23

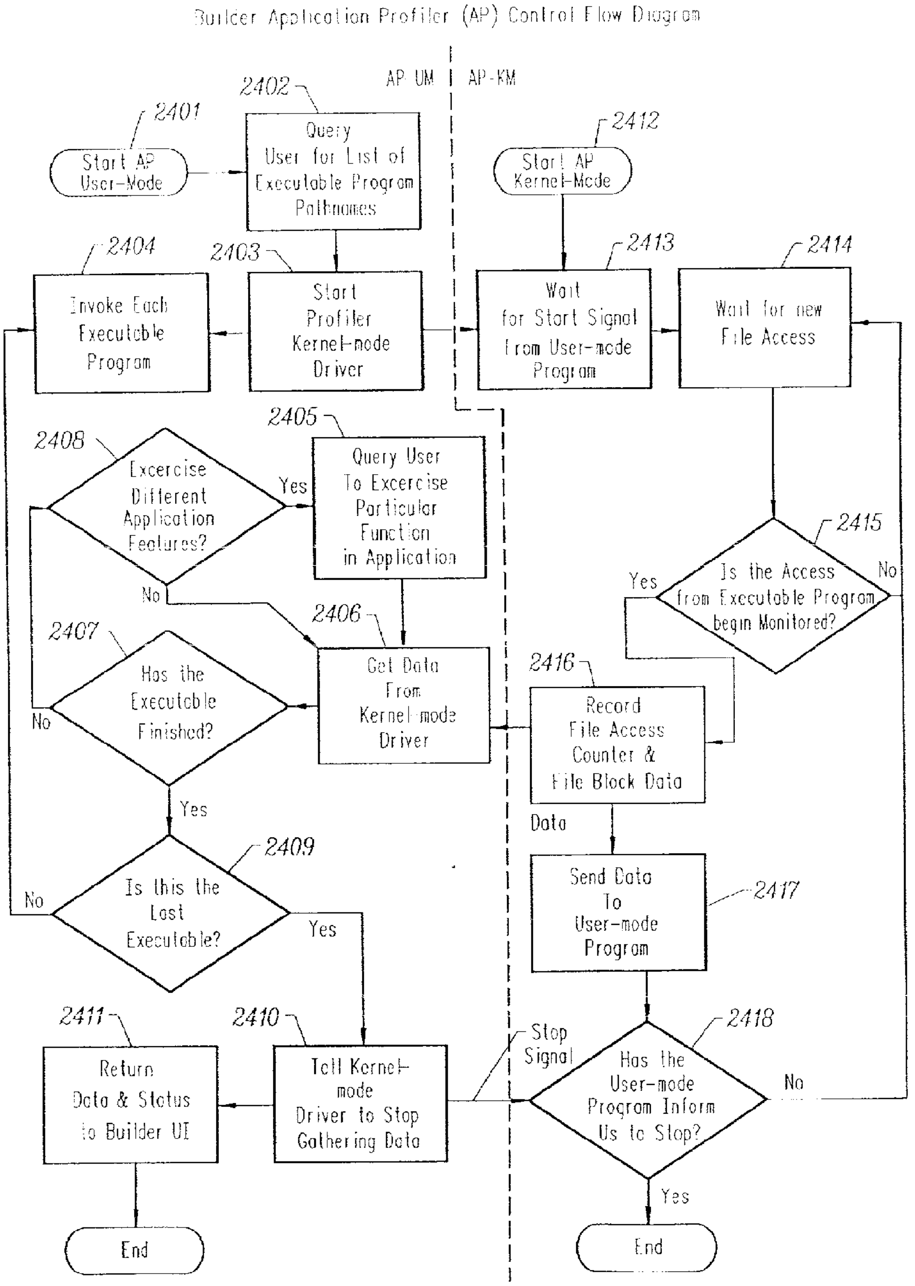

Builder Application Profiler (AP) Control Flow Diagram
AP UM
AP-KM
2401
Start AP User-Mode
2402
Query User for List of Executable Program Pathnames
2403
Start Profiler Kernel-mode Driver
2404
Invoke Each Executable Program
2405
Query User To Excercise Particular Function in Application
2408
Excercise Different Application Features?
Yes
No
2406
Get Data From Kernel-mode Driver
2407
Has the Executable Finished?
No
Yes
2409
Is this the Last Executable?
No
Yes
2410
Tell Kernel-mode Driver to Stop Gathering Data
2411
Return Data & Status to Builder UI
End
Stop Signal
2412
Start AP Kernel-Mode
2413
Wait for Start Signal From User-mode Program
2414
Wait for new File Access
2415
Is the Access from Executable Program begin Monitored?
Yes
No
2416
Record File Access Counter & File Block Data
Data
2417
Send Data To User-mode Program
2418
Has the User-mode Program Inform Us to Stop?
No
Yes
End

FIG. 24

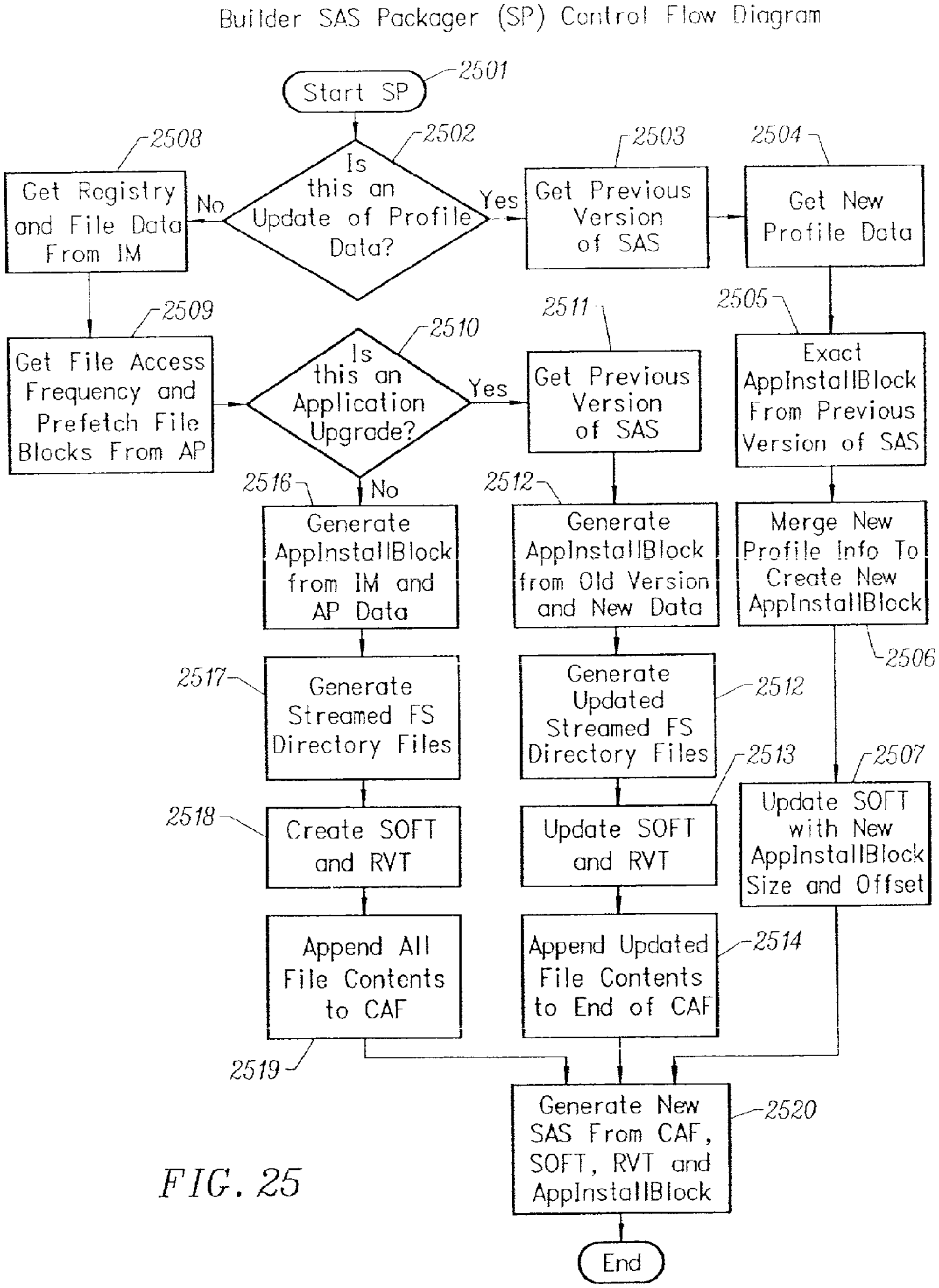

Builder SAS Packager (SP) Control Flow Diagram
Start SP
2501
Is this an Update of Profile Data?
2502
Yes
No
Get Previous Version of SAS
2503
Get New Profile Data
2504
Get Registry and File Data From IM
2508
Get File Access Frequency and Prefetch File Blocks From AP
2509
Is this an Application Upgrade?
2510
Yes
No
Get Previous Version of SAS
2511
Exact AppInstallBlock From Previous Version of SAS
2505
Generate AppInstallBlock from IM and AP Data
2516
Generate AppInstallBlock from Old Version and New Data
2512
Merge New Profile Info To Create New AppInstallBlock
2506
Generate Streamed FS Directory Files
2517
Generate Updated Streamed FS Directory Files
2512
Create SOFT and RVT
2518
Update SOFT and RVT
2513
Update SOFT with New AppInstallBlock Size and Offset
2507
Append All File Contents to CAF
2519
Append Updated File Contents to End of CAF
2514
Generate New SAS From CAF, SOFT, RVT and AppInstallBlock
2520
End

FIG. 25

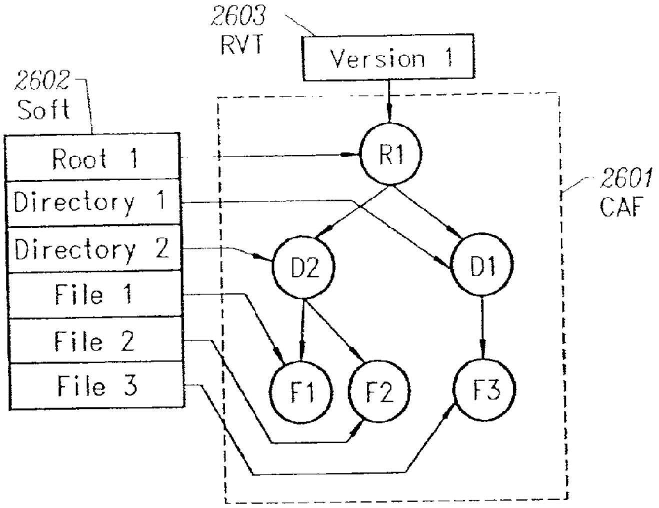


FIG. 26A

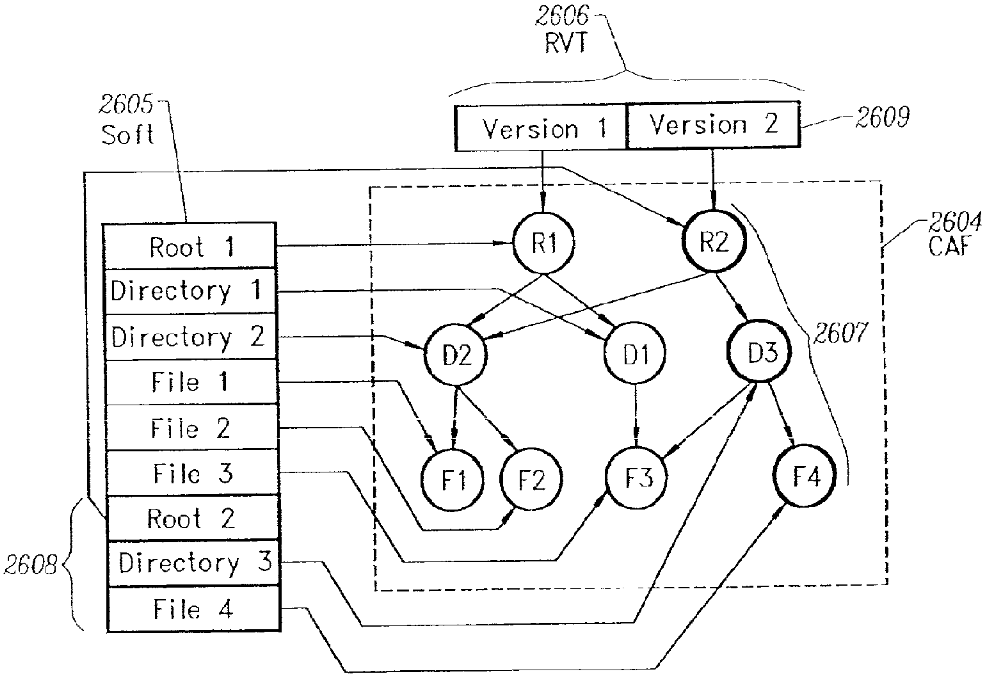


FIG. 26B

Streamed Application Set (SAS) Builder Data Flow Diagram

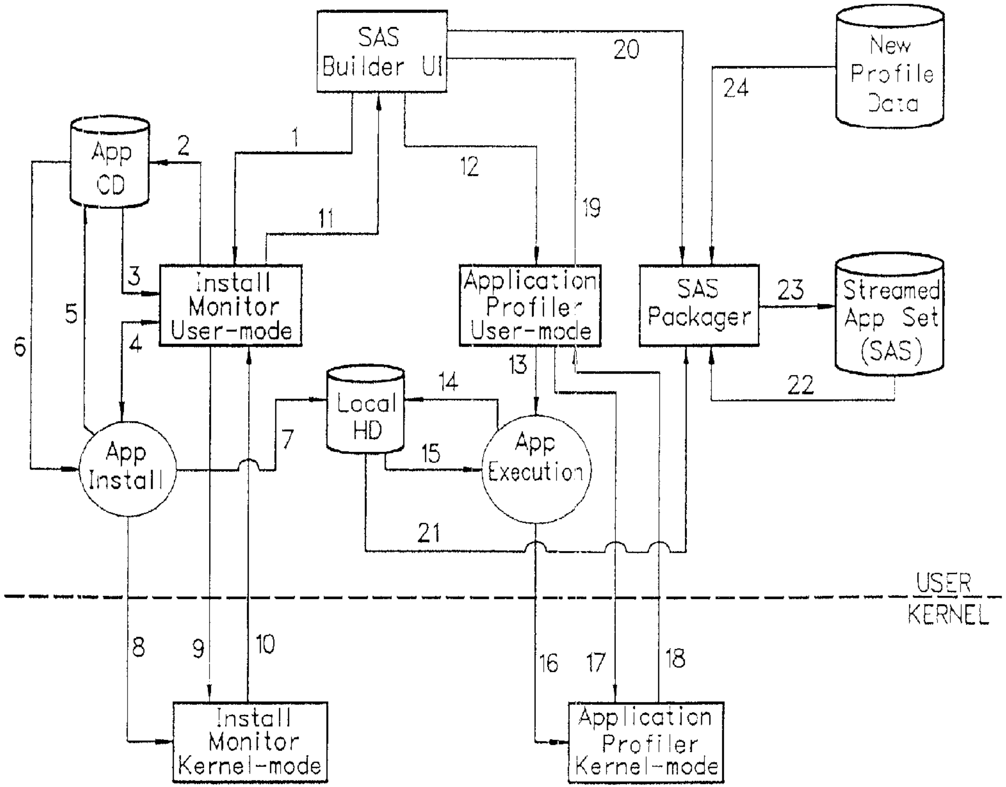


FIG. 27

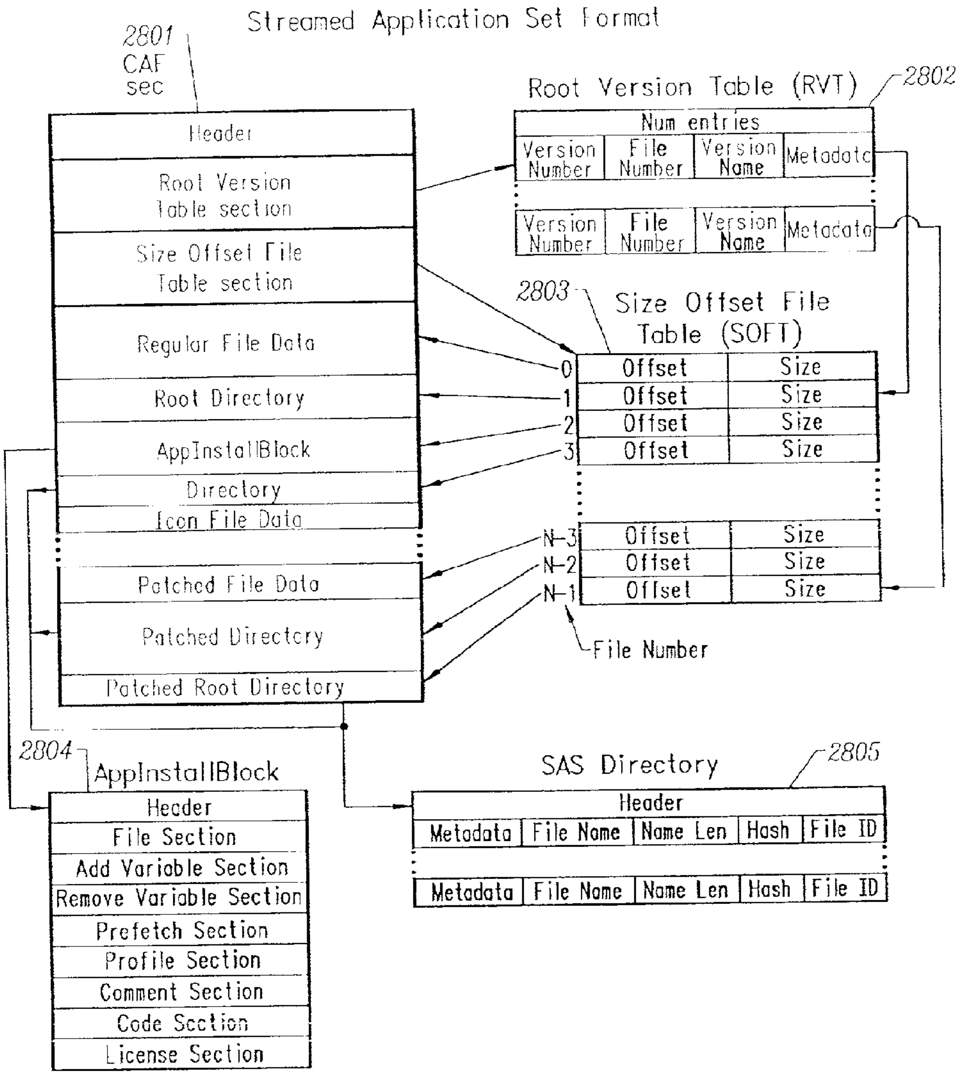

Streamed Application Set Format
2801
CAF sec
Header
Root Version Table section
Size Offset File Table section
Regular File Data
Root Directory
AppInstallBlock
Directory
Icon File Data
Patched File Data
Patched Directory
Patched Root Directory
Root Version Table (RVT)
2802
Num entries
Version Number
File Number
Version Name
Metadata
Version Number
File Number
Version Name
Metadata
2803
Size Offset File Table (SOFT)
0
1
2
3
Offset
Size
N-3
N-2
N-1
File Number
2804
AppInstallBlock
Header
File Section
Add Variable Section
Remove Variable Section
Prefetch Section
Profile Section
Comment Section
Code Section
License Section
SAS Directory
2805
Header
Metadata
File Name
Name Len
Hash
File ID
Metadata
File Name
Name Len
Hash
File ID

FIG. 28

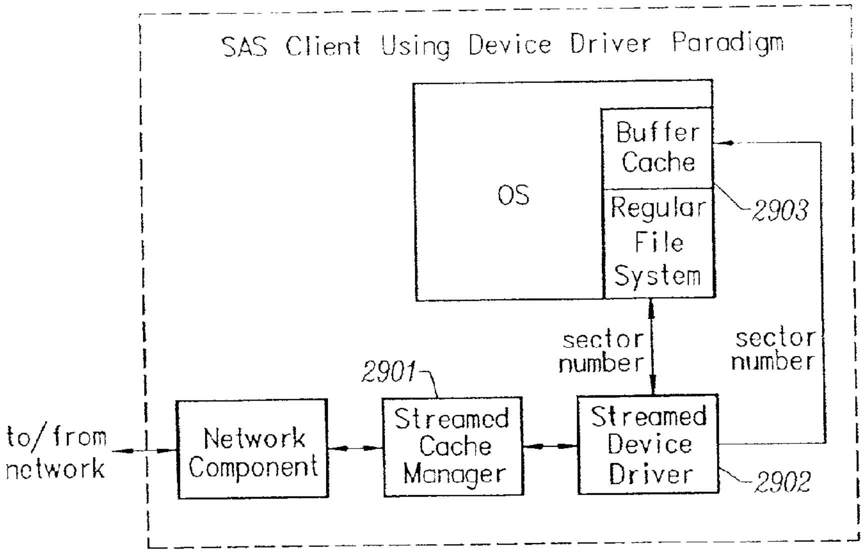

SAS Client Using Device Driver Paradigm
OS
Buffer Cache
Regular File System
2903
sector number
sector number
2901
to/from network
Network Component
Streamed Cache Manager
Streamed Device Driver
2902

FIG. 29

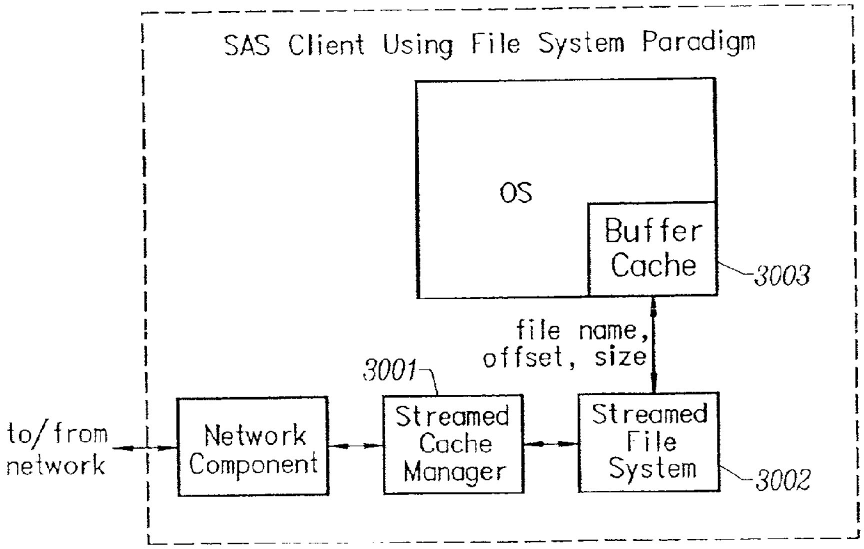

SAS Client Using File System Paradigm
OS
Buffer Cache
3003
file name, offset, size
3001
to/from network
Network Component
Streamed Cache Manager
Streamed File System
3002

FIG. 30

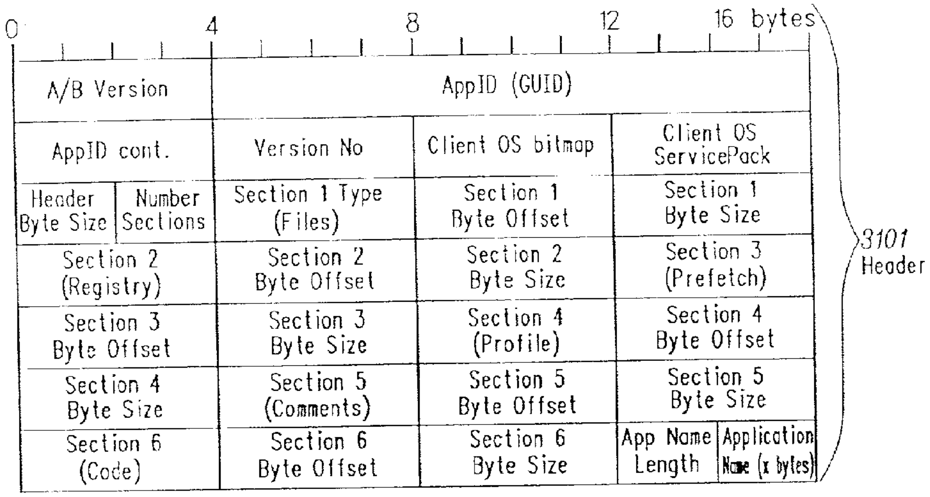


FIG. 31A

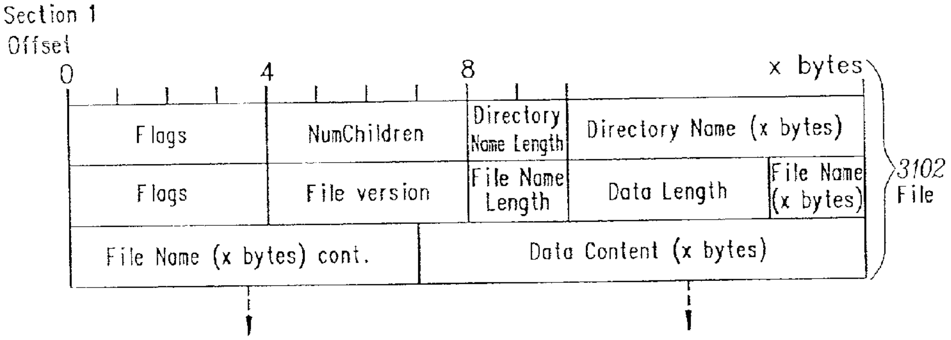


FIG. 31B

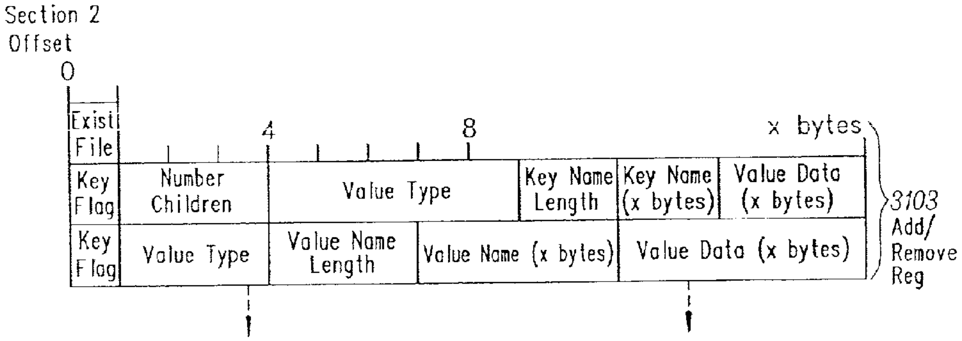


FIG. 31C

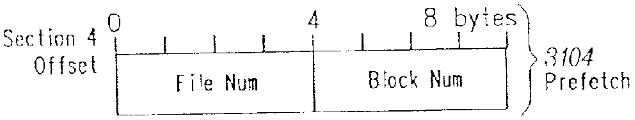


FIG. 31D

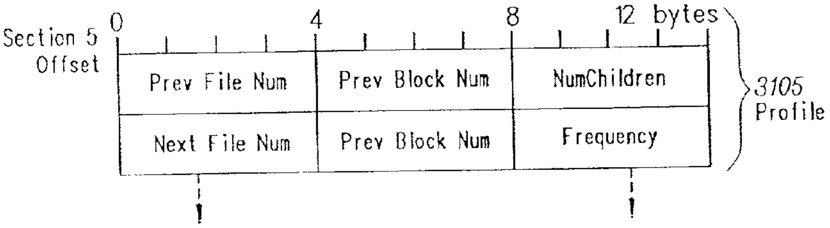


FIG. 31E

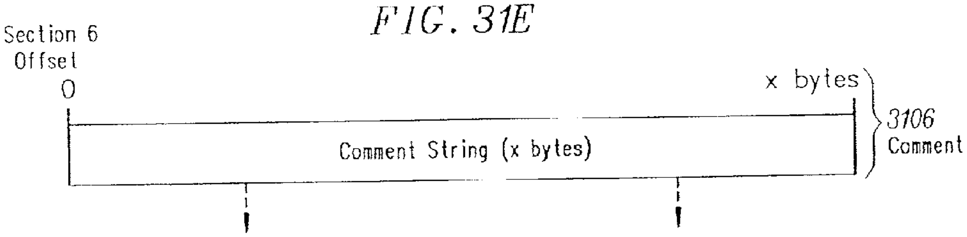


FIG. 31F

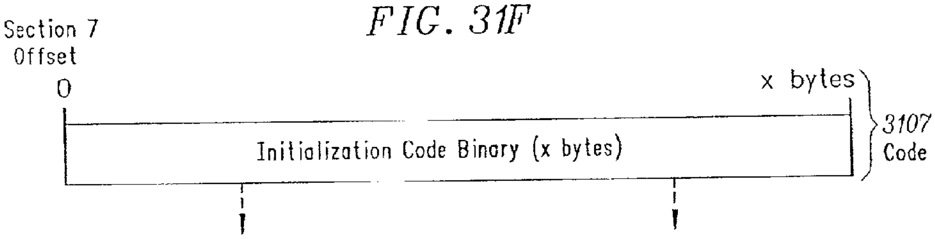


FIG. 31G

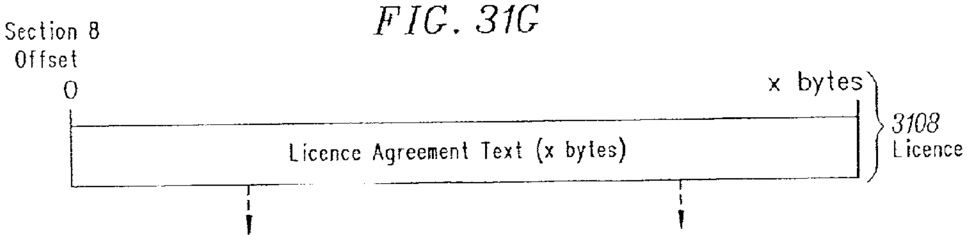


FIG. 31H

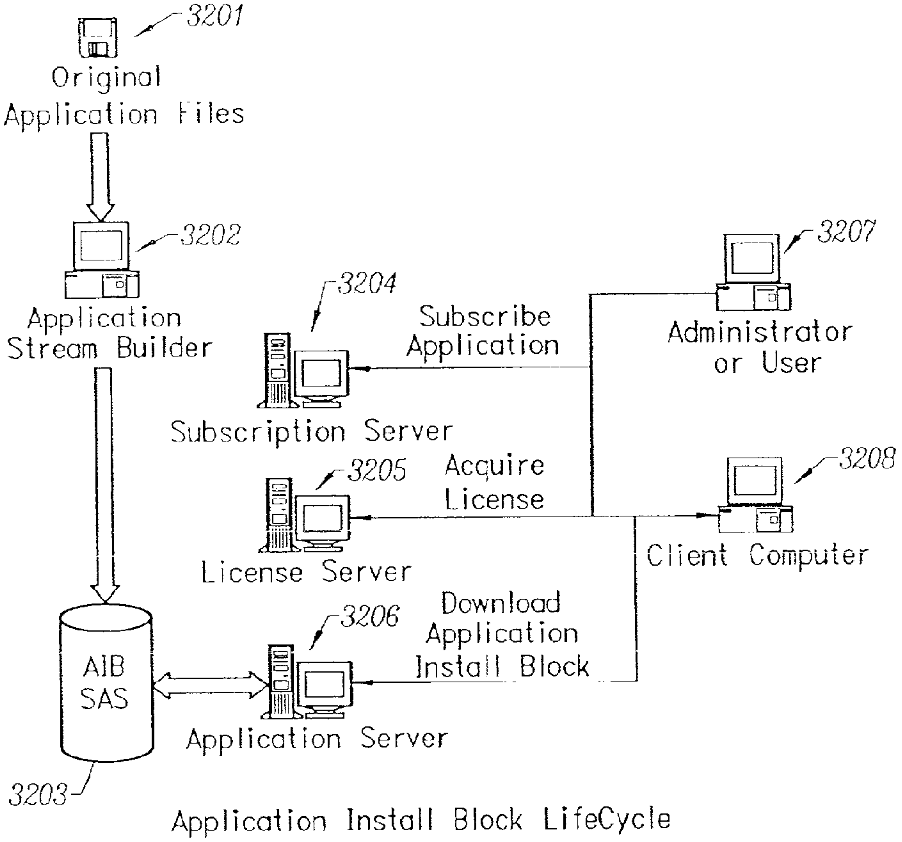

3201
Original Application Files
3202
Application Stream Builder
3204
Subscribe Application
Subscription Server
3207
Administrator or User
3205
Acquire License
License Server
3208
Client Computer
3206
Download Application Install Block
AIB SAS
Application Server
3203
Application Install Block LifeCycle

FIG. 32

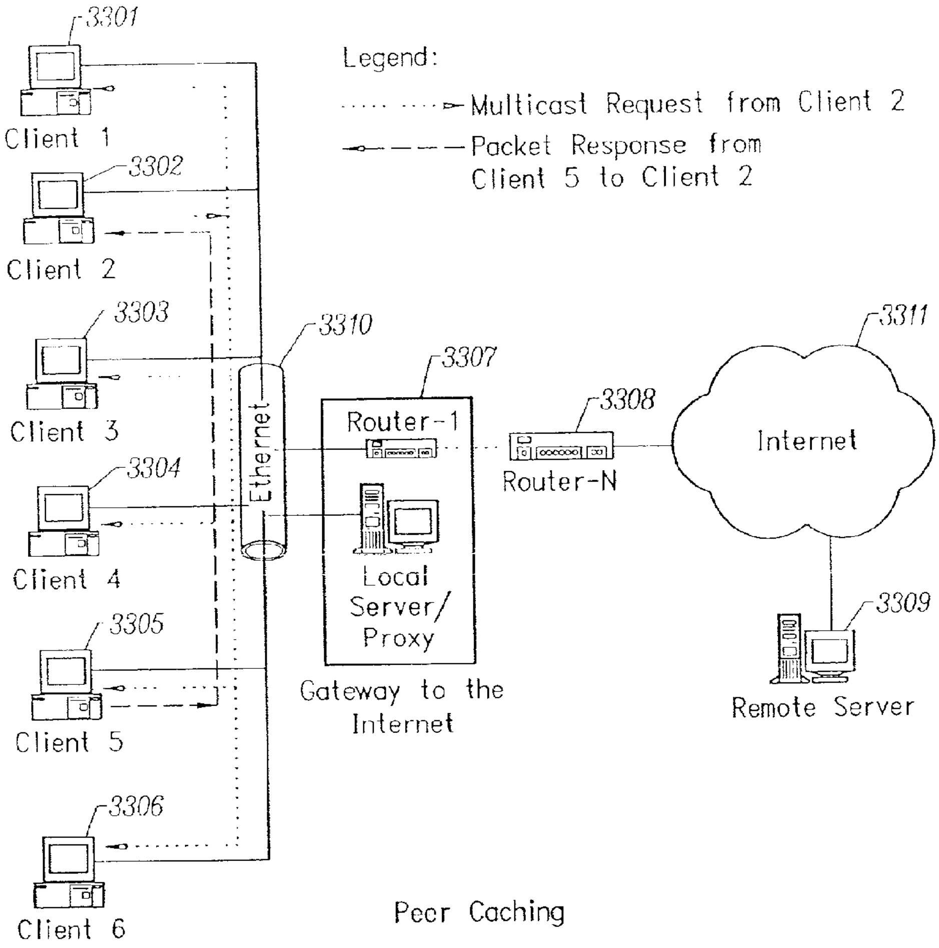

3301
Client 1
3302
Client 2
3303
Client 3
3304
Client 4
3305
Client 5
3306
Client 6
Legend:
Multicast Request from Client 2
Packet Response from Client 5 to Client 2
3310
Ethernet
3307
Router-1
Local Server/ Proxy
Gateway to the Internet
3308
Router-N
3311
Internet
3309
Remote Server
Peer Caching

FIG. 33

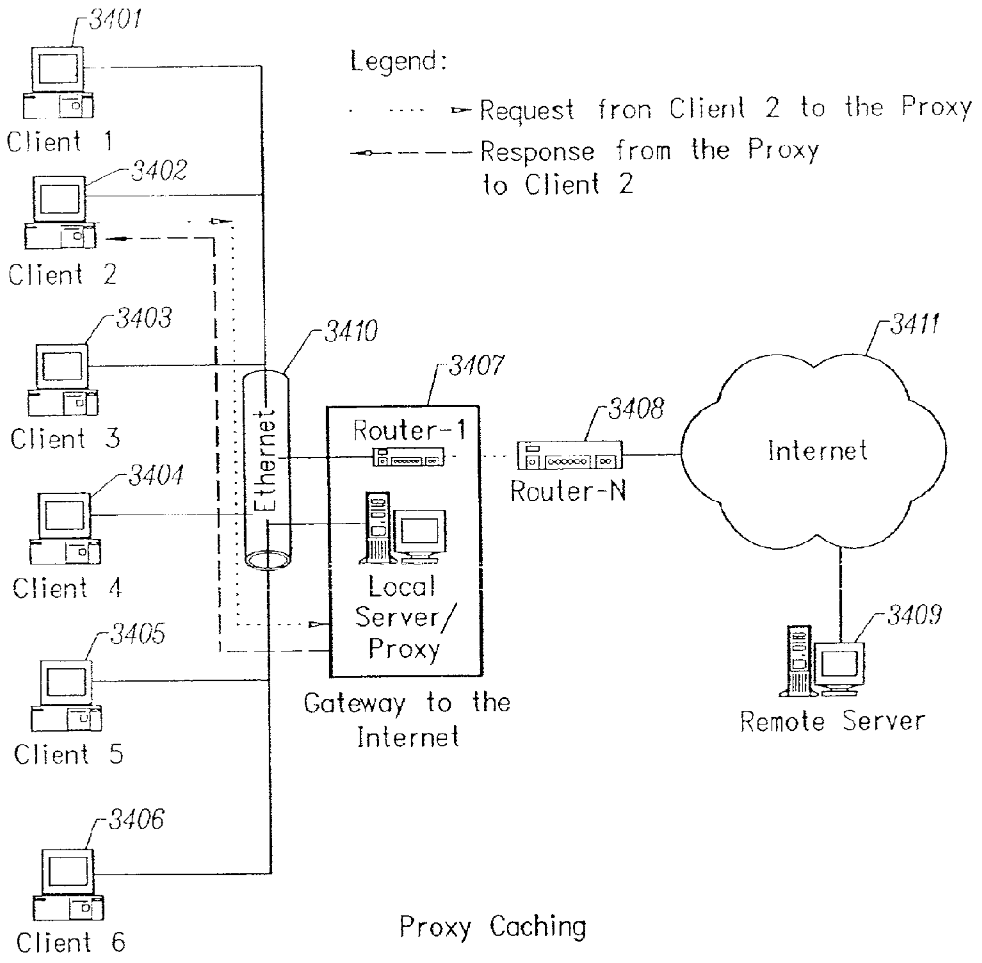
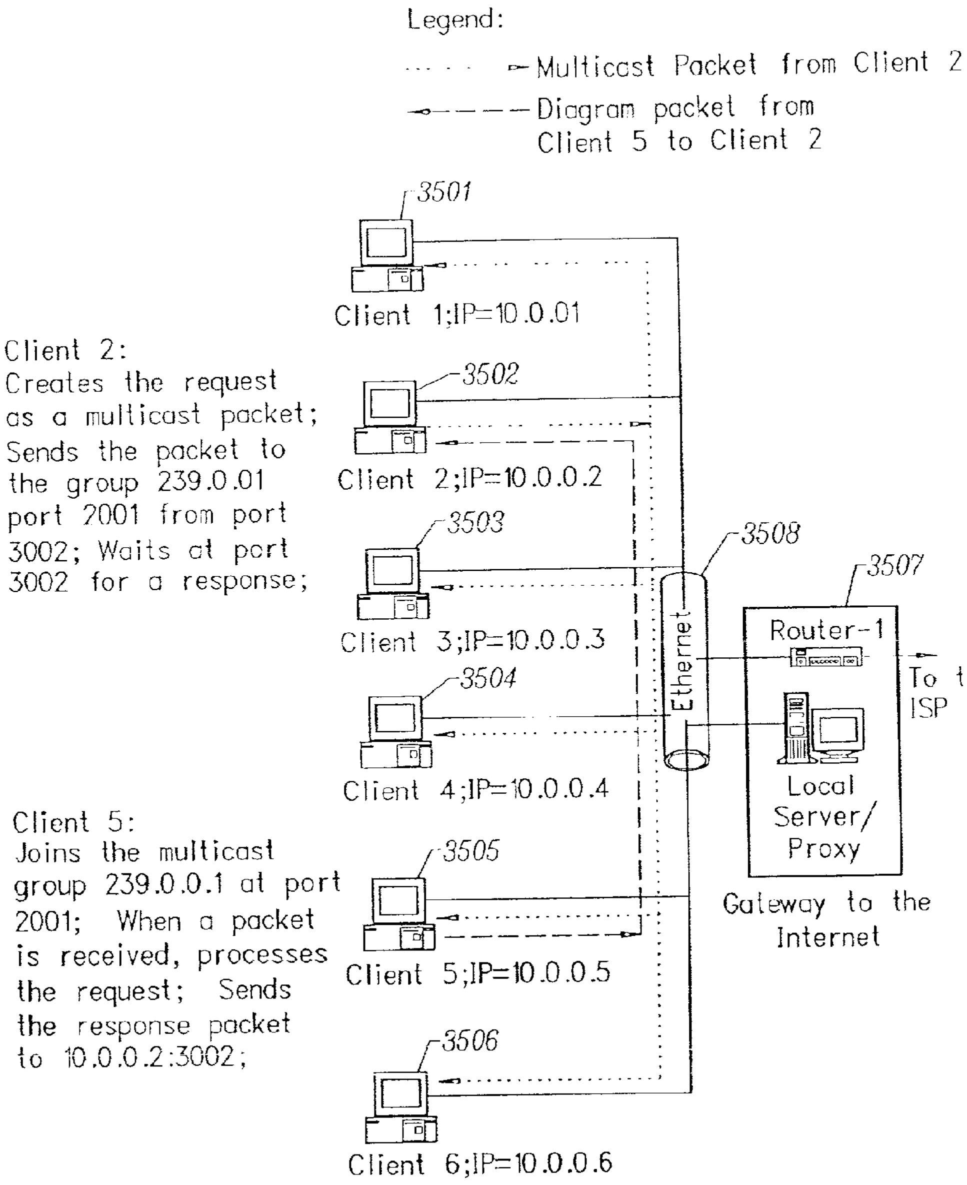
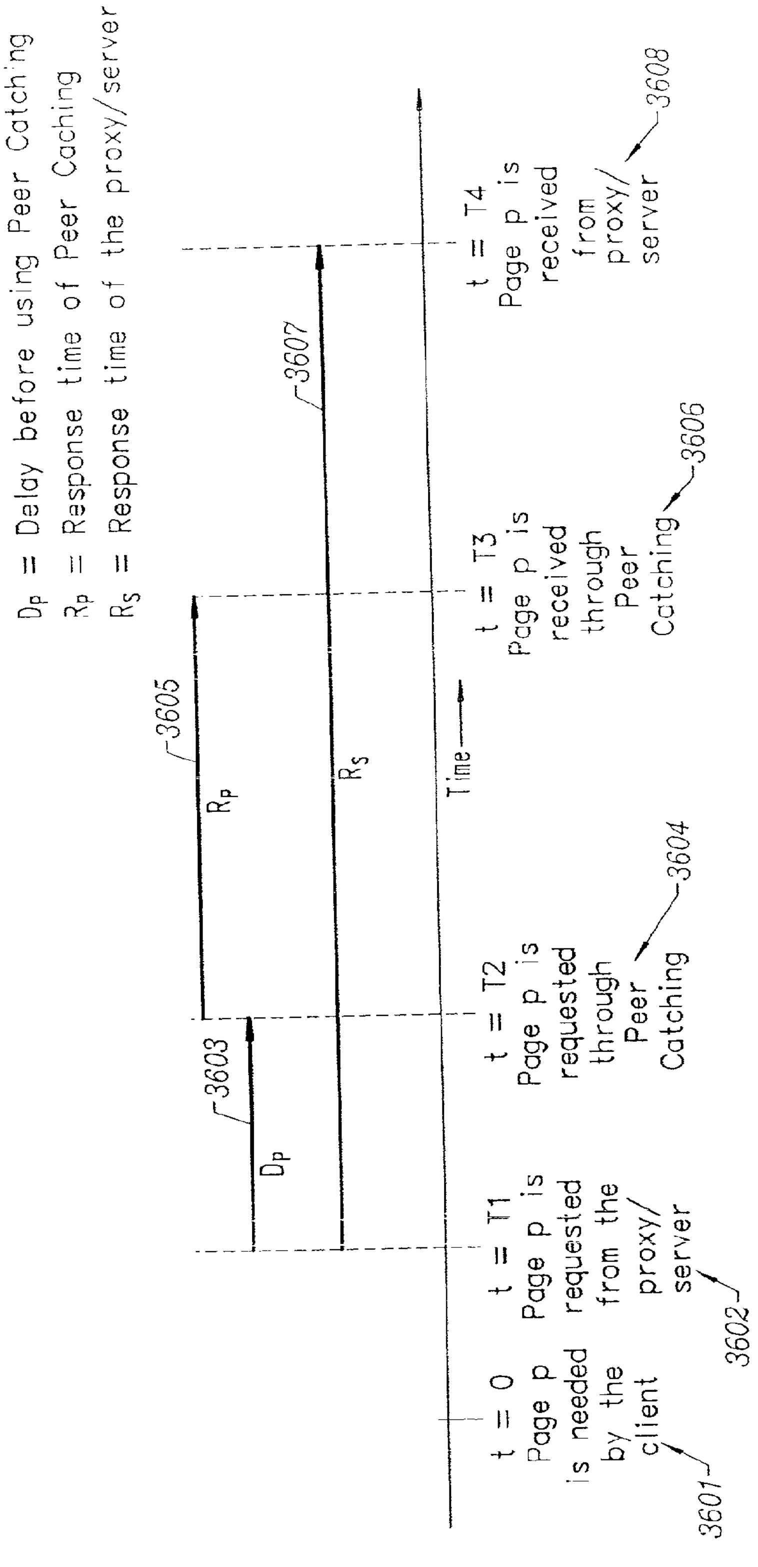

3401
Client 1
3402
Client 2
3403
Client 3
3404
Client 4
3405
Client 5
3406
Client 6
Legend:
Request fron Client 2 to the Proxy
Response from the Proxy to Client 2
3410
Ethernet
3407
Router-1
Local Server/ Proxy
Gateway to the Internet
3408
Router-N
3411
Internet
3409
Remote Server
Proxy Caching

FIG. 34


Legend:
Multicast Packet from Client 2
Diagram packet from Client 5 to Client 2
3501
Client 1;IP=10.0.01
Client 2:
Creates the request as a multicast packet; Sends the packet to the group 239.0.01 port 2001 from port 3002; Waits at port 3002 for a response;
3502
Client 2;IP=10.0.0.2
3503
Client 3;IP=10.0.0.3
3504
Client 4;IP=10.0.0.4
3508
Ethernet
3507
Router-1
To the ISP
Local Server/ Proxy
Gateway to the Internet
Client 5:
Joins the multicast group 239.0.0.1 at port 2001; When a packet is received, processes the request; Sends the response packet to 10.0.0.2:3002;
3505
Client 5;IP=10.0.0.5
3506
Client 6;IP=10.0.0.6

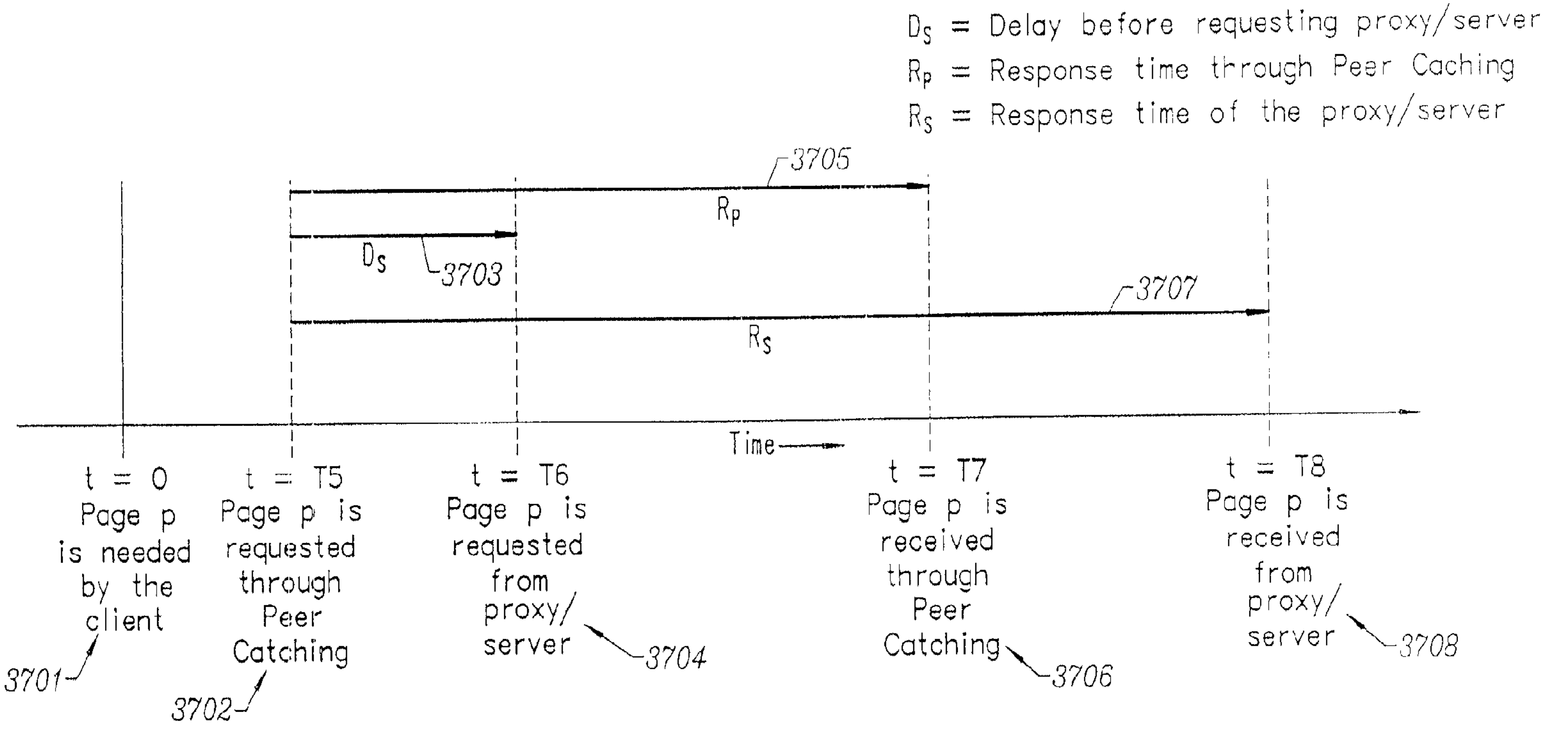
Milticast within a LAN and Packet Protocol

FIG. 35


D_P = Delay before using Peer Catching
R_P = Response time of Peer Caching
R_S = Response time of the proxy/server
t = 0 Page p is needed by the client
3601
t = T1 Page p is requested from the proxy/ server
3602
D_P
3603
t = T2 Page p is requested through Peer Catching
3604
R_P
3605
Time
R_S
t = T3 Page p is received through Peer Catching
3606
3607
t = T4 Page p is received from proxy/ server
3608

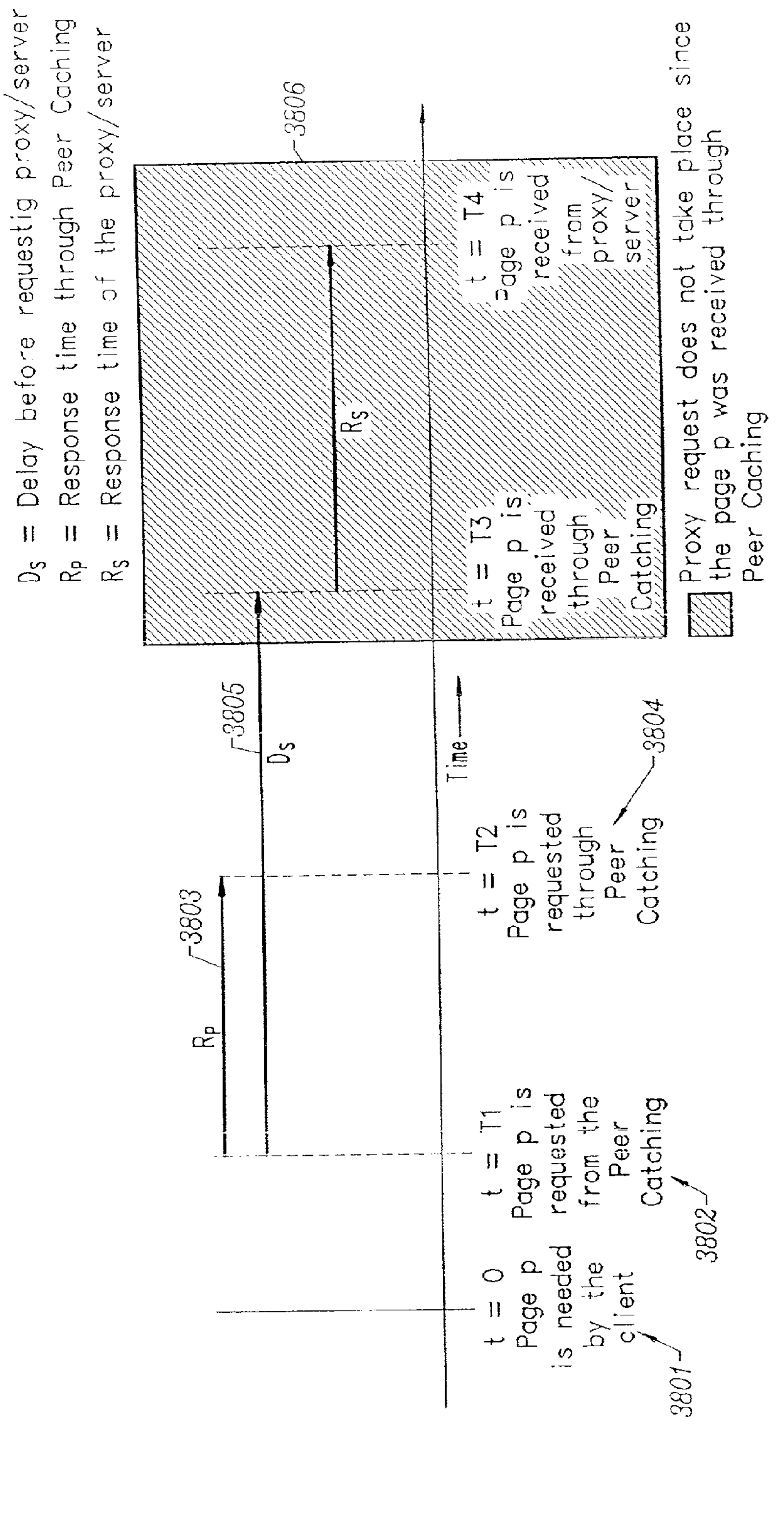
Concurrent Requesting - Proxy First

FIG. 36



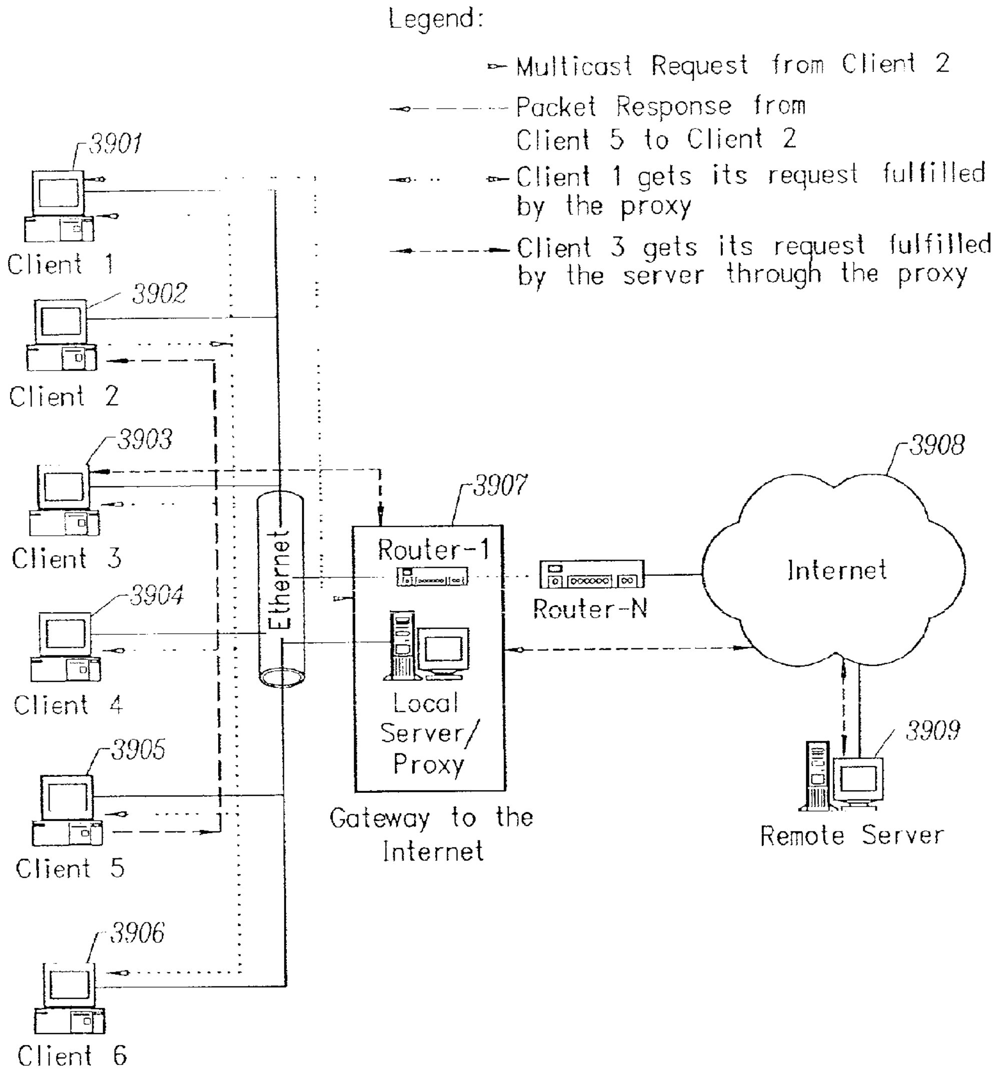
Concurrent Requesting - Peer Catching First

FIG. 37



Concurrent Requesting - Peer Caching Only

FIG. 38


Legend:
Multicast Request from Client 2
Packet Response from Client 5 to Client 2
Client 1 gets its request fulfilled by the proxy
Client 3 gets its request fulfilled by the server through the proxy
3901
Client 1
3902
Client 2
3903
Client 3
3904
Client 4
3905
Client 5
3906
Client 6
Ethernet
3907
Router-1
Local Server/ Proxy
Gateway to the Internet
Router-N
3908
Internet
3909
Remote Server

Client-Server System with Peer and Proxy Caching

FIG. 39

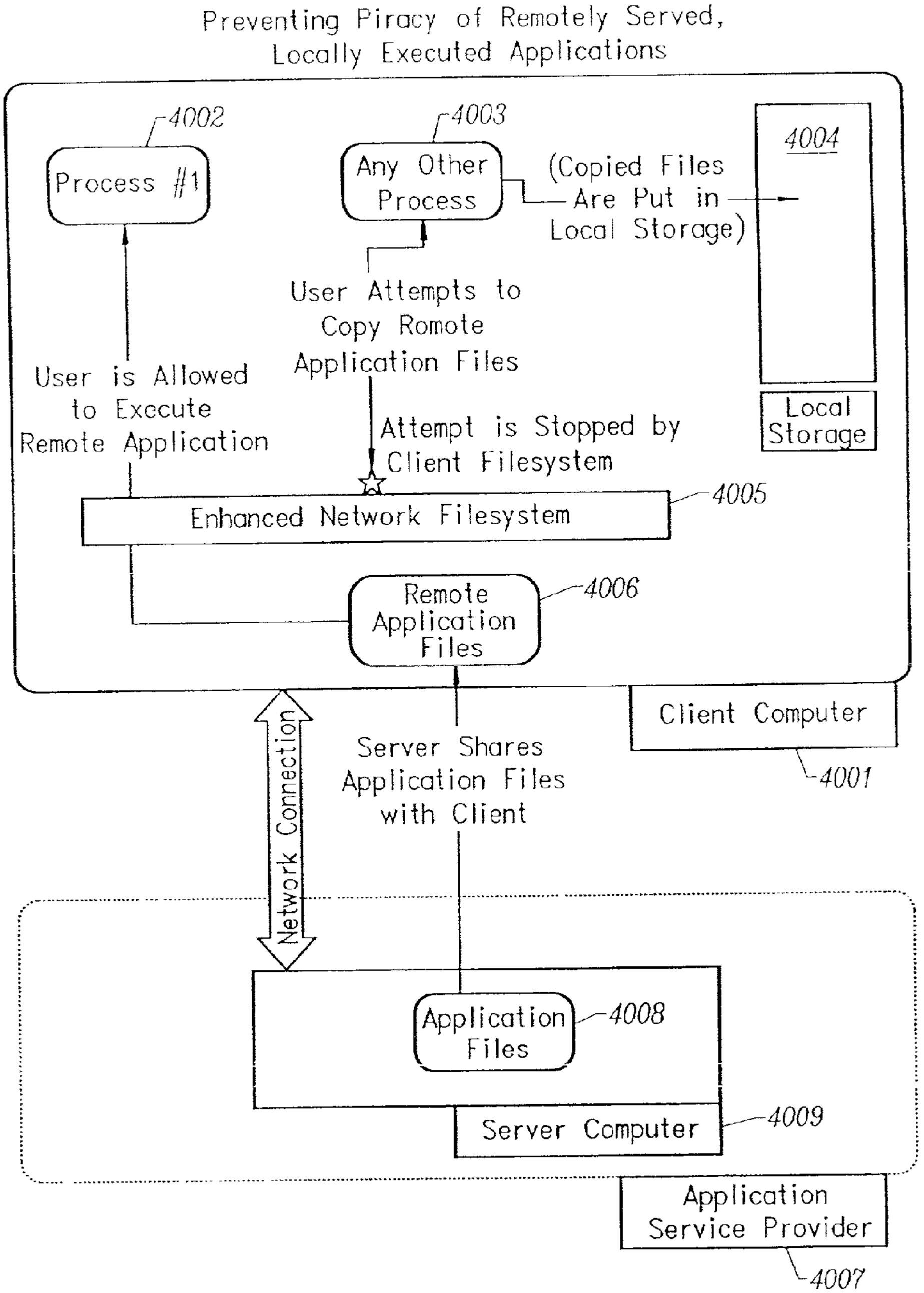

Preventing Piracy of Remotely Served, Locally Executed Applications
4002
Process #1
4003
Any Other Process
(Copied Files Are Put in Local Storage)
4004
User Attempts to Copy Romote Application Files
User is Allowed to Execute Remote Application
Attempt is Stopped by Client Filesystem
Local Storage
Enhanced Network Filesystem
4005
Remote Application Files
4006
Client Computer
4001
Network Connection
Server Shares Application Files with Client
Application Files
4008
Server Computer
4009
Application Service Provider
4007

FIG. 40

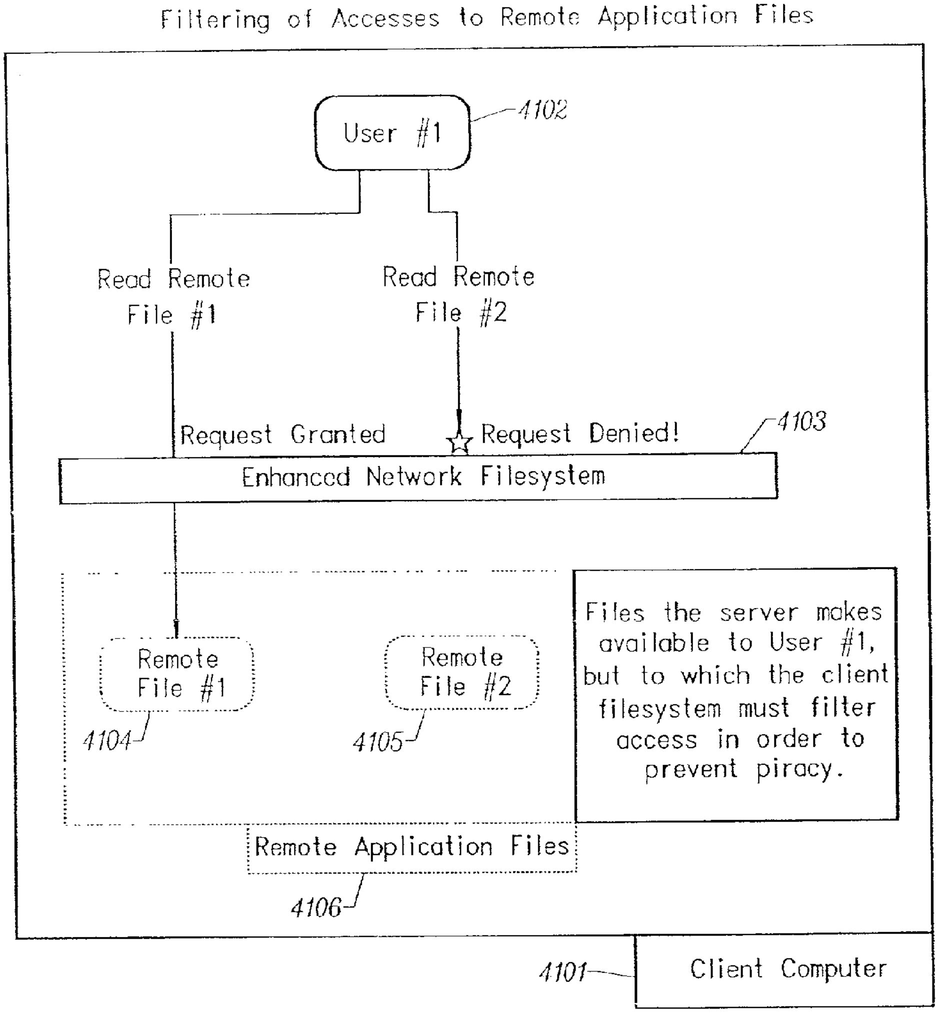

Filtering of Accesses to Remote Application Files
User #1
4102
Read Remote File #1
Read Remote File #2
Request Granted
Request Denied!
4103
Enhanced Network Filesystem
Remote File #1
Remote File #2
4104
4105
Files the server makes available to User #1, but to which the client filesystem must filter access in order to prevent piracy.
Remote Application Files
4106
4101
Client Computer

FIG. 41

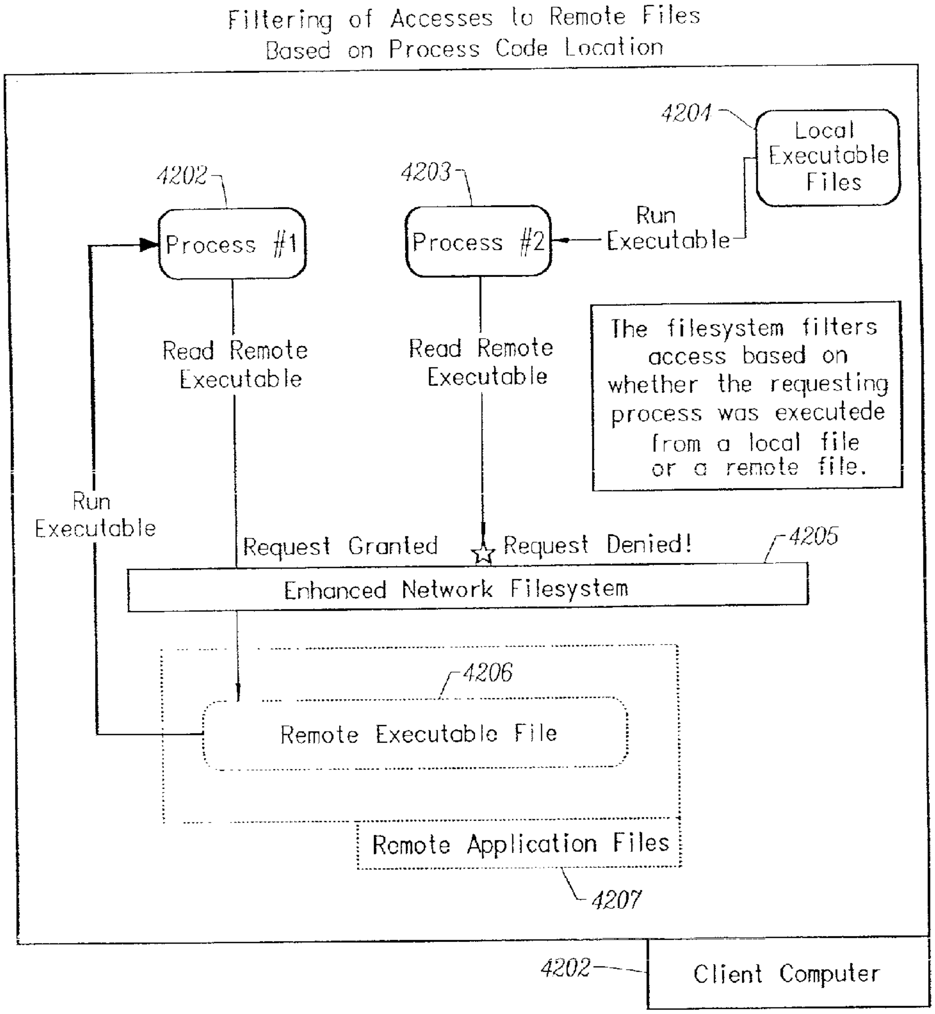

Filtering of Accesses to Remote Files
Based on Process Code Location
4204
Local Executable Files
4202
4203
Process #1
Process #2
Run Executable
The filesystem filters access based on whether the requesting process was executede from a local file or a remote file.
Read Remote Executable
Read Remote Executable
Run Executable
Request Granted
Request Denied!
4205
Enhanced Network Filesystem
4206
Remote Executable File
Remote Application Files
4207
4202
Client Computer

FIG. 42

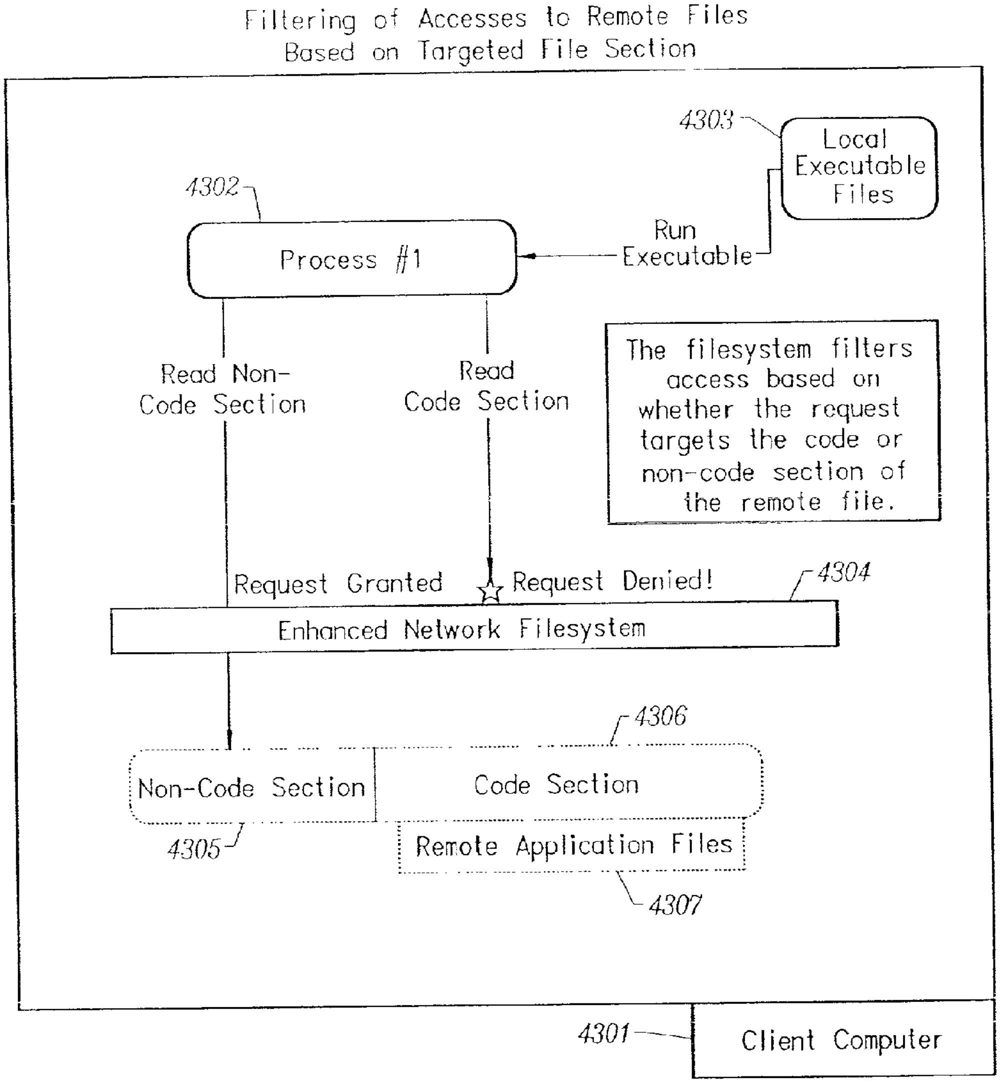

Filtering of Accesses to Remote Files
Based on Targeted File Section
4303
Local Executable Files
4302
Process #1
Run Executable
Read Non-Code Section
Read Code Section
The filesystem filters access based on whether the request targets the code or non-code section of the remote file.
Request Granted
Request Denied!
4304
Enhanced Network Filesystem
4306
Non-Code Section
Code Section
4305
Remote Application Files
4307
4301
Client Computer

FIG. 43

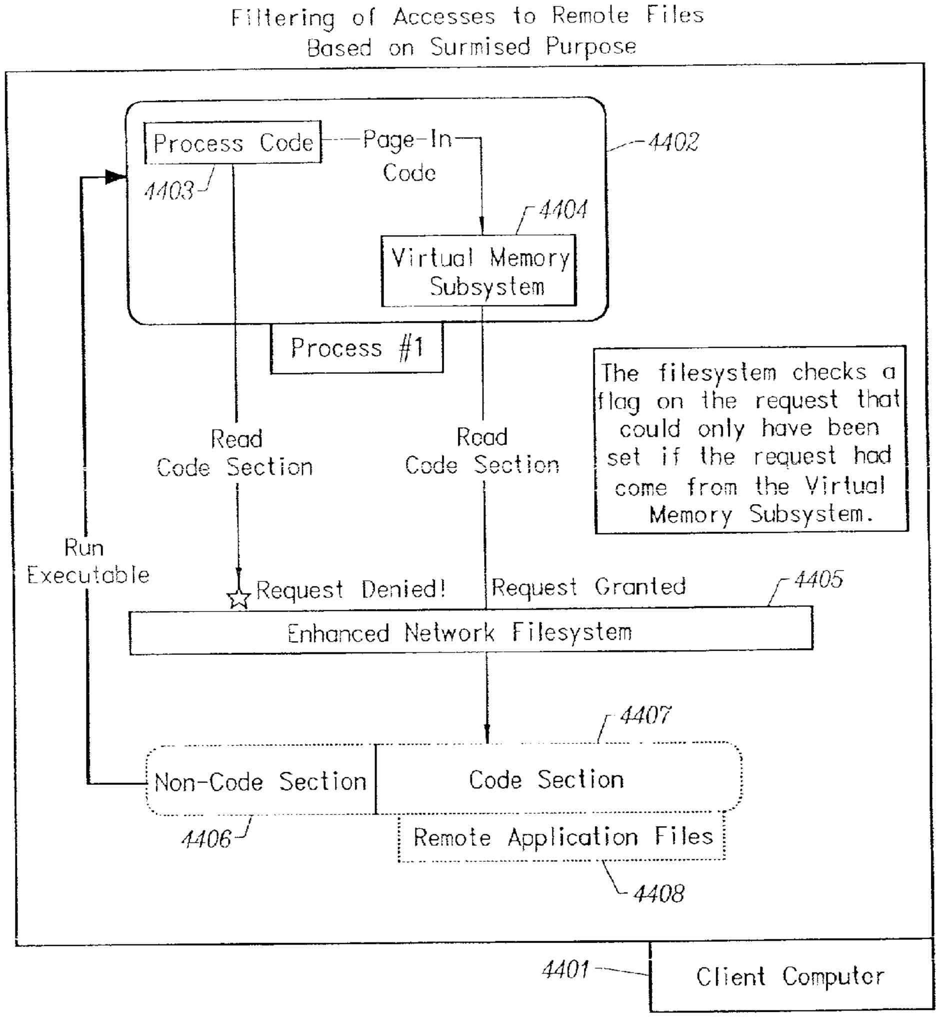

Filtering of Accesses to Remote Files
Based on Surmised Purpose
Process Code
Page-In Code
4402
4403
4404
Virtual Memory Subsystem
Process #1
The filesystem checks a flag on the request that could only have been set if the request had come from the Virtual Memory Subsystem.
Read Code Section
Read Code Section
Run Executable
Request Denied!
Request Granted
4405
Enhanced Network Filesystem
4407
Non-Code Section
Code Section
4406
Remote Application Files
4408
4401
Client Computer

FIG. 44

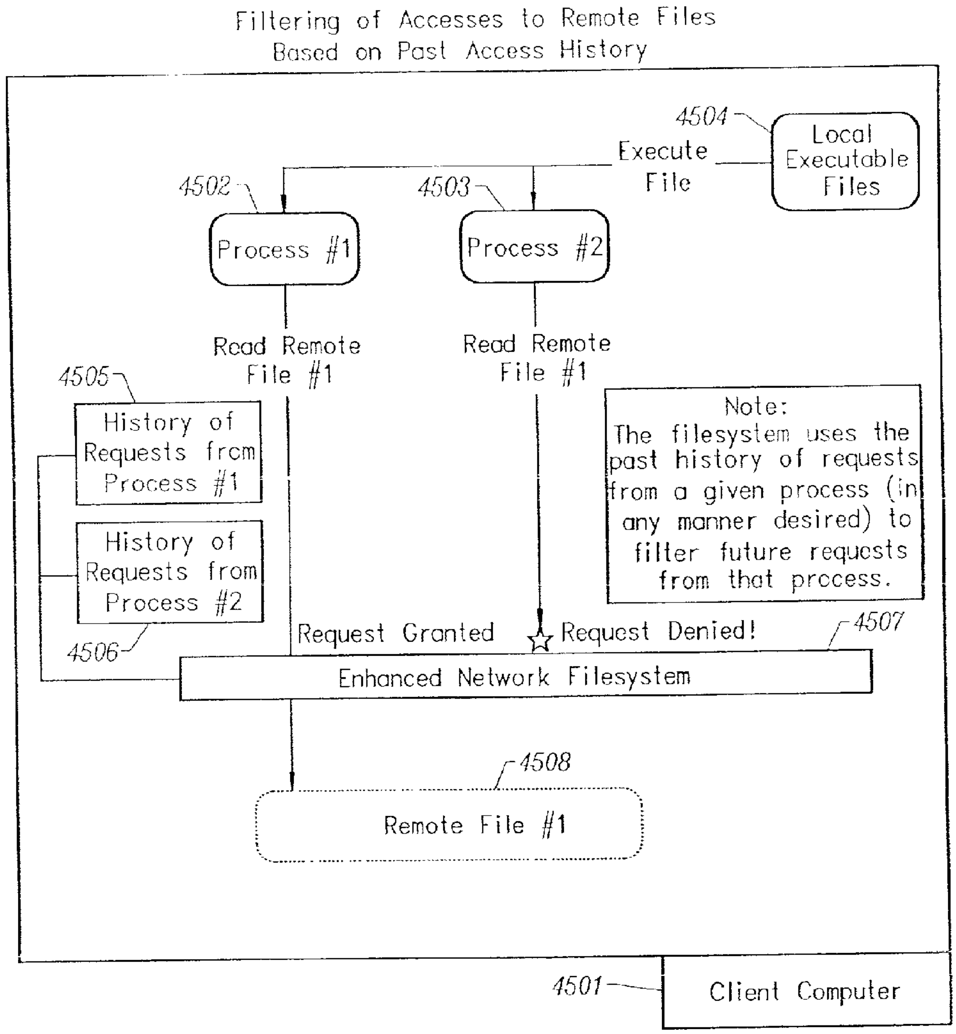

Filtering of Accesses to Remote Files
Based on Past Access History
4504
Local
Executable
Files
Execute
File
4502
4503
Process #1
Process #2
Read Remote
File #1
Read Remote
File #1
4505
History of
Requests from
Process #1
History of
Requests from
Process #2
4506
Note:
The filesystem uses the
past history of requests
from a given process (in
any manner desired) to
filter future requests
from that process.
Request Granted
Request Denied!
4507
Enhanced Network Filesystem
4508
Remote File #1
4501
Client Computer

FIG. 45

CLIENT-SIDE PERFORMANCE OPTIMIZATION SYSTEM FOR STREAMED APPLICATIONS

CROSS REFERENCE TO RELATED APPLICATIONS

[0001] This application Claims benefit of U.S. Provisional Patent Application Ser. No. 60/246,384, filed on Nov. 6, 2000 (OTI.2000.0).

BACKGROUND OF THE INVENTION

[0002] 1. Technical Field

[0003] The invention relates to the streaming of computer program object code across a network in a computer environment. More particularly, the invention relates to client-side data retrieval, storage, and execution performance optimization techniques for computer program object code and other related data streamed across a network from a server.

[0004] 2. Description of the Prior Art

[0005] Retail sales models of computer application programs are fairly straight forward. The consumer either purchases the application program from a retailer that is either a brick and mortar or an ecommerce entity. The product is delivered to the consumer in a shrink-wrap form.

[0006] The consumer installs the program from a floppy disk or a CD-ROM included in the packaging. A serial number is generally provided that must be entered at installation or the first time the program is run. Other approaches require that the CD-ROM be present whenever the program is run. However, CD-ROMs are easily copied using common CDR technology.

[0007] Another approach is for the consumer to effectuate the purchase through an ecommerce entity. The application program is downloaded in its entirety to the consumer across the Internet. The consumer is emailed a serial number that is required to run the program. The consumer enters the serial number at the time the program is installed or the first time the program is run.

[0008] Once the application program is installed on a machine, it resides on the machine, occupying precious hard disk space, until It is physically removed. The installer portion of the program can also be installed on a server along with the installation files. Users within an intranet can install the program from the server, across the network, onto their machines. The program is a full installation of the program and resides on the user's machine until it is manually removed.

[0009] Trial versions of programs are also available online that are a partial or full installation of the application program. The program executes normally for a preset time period. At the end of the time period, the consumer is told that he must purchase the program and execution is terminated. The drawback to this approach is that there is an easy way for the consumer to fool the program. The consumer simply uninstalls the program and then reinstalls it, thereby restarting the time period.

[0010] Additionally, piracy problems arise once the application program is resident on the consumer's computer. Serial numbers for programs are easily obtained across the Internet. Software companies lose billions of dollars a year in revenue because of this type of piracy.

[0011] The above approaches fail to adequately protect software companies' revenue stream. These approaches also require the consumer to install a program that resides indefinitely on the consumer's hard disk, occupying valuable space even though the consumer may use the program infrequently.

[0012] The enterprise arena allows Application Service Providers (ASP) to provide browser-based implementations such as Tarantella offered by Santa Cruz Operation, Inc. in Santa Cruz, Calif. and Metaframe offered by Citrix Systems Inc. of Fort Lauderdale, Fla. A remote application portal site allows the user to click on an application in his browser to execute the application. The application runs on the portal site and GUI interfaces such as display, keystrokes and mouse clicks are transferred over the wire. The access to the program is password protected. This approach allows the provider to create an audit trail and to track the use of an application program.

[0013] AppStream Inc. of Palo Alto, Calif. uses Java code streamlets to provide streaming applications to the user. The system partitions a Web application program into Java streamlets. Java streamlets are then streamed to the user's computer on an as-needed basis. The application runs on the user's computer, but is accessed through the user's network browser.

[0014] The drawback to the browser-based approaches is that the user is forced to work within his network browser, thereby adding another layer of complexity. The browser or Java program manages the application program's run-time environment. The user loses the experience that the software manufacturer had originally intended for its product including features such as application invocation based on file extension associations.

[0015] It would be advantageous to provide a client-side performance optimization system for streamed applications that enables a client system to efficiently stream and execute application programs that are remotely served from a server. It would further be advantageous to provide a client-side performance optimization system for streamed applications that easily integrates into the client system's operating system.

SUMMARY OF THE INVENTION

[0016] The invention provides a client-side performance optimization system for streamed applications. The system enables a client system to efficiently stream and execute application programs that are remotely served from a server. In addition, the invention provides a system that that easily integrates into the client system's operating system.

[0017] The invention provides several approaches for fulfilling client-side application code and data file requests for streamed applications. A streaming file system or file driver is installed on the client system that receives and fulfills application code and data requests.

[0018] One approach installs an application streaming file system on the client machine that appears to contain the installed application. The application streaming file system receive all requests for code or data that are part of the application and satisfies requests for application code or data by retrieving it from its persistent cache or by retrieving it directly from the streaming application server. Code or data retrieved from the server is placed in the cache for reuse.

[0019] Another approach installs a kernel-mode streaming file system driver and a user-mode client. Requests made to the streaming file system driver are directed to the user-mode client which handles the streams from the application streaming server or persistent cache and sends the results back to the driver.

[0020] Yet another approach is comprised of a streaming block driver on the client system. It appears as a physical disk to the native file system already installed on the client operating system. The driver receives requests for application code and data block reads and writes and satisfies the requests from the persistent cache or the streaming application server.

[0021] A final approach adds a disk driver and a user mode client on the client system. The disk driver sends program code and data requests to the user-mode client which satisfies them out of the persistent cache or by going to the streaming application server.

[0022] The persistent cache may be encrypted with a key not permanently stored on the client to prevent unauthorized use or duplication of application code or data. The key is sent to the client by the streaming application server upon application startup and is not stored in the application's persistent storage area.

[0023] The client can initiate the prefetching of application code and data to improve interactive application performance. The client software examines code and data requests and consults the contents of the persistent cache as well as historic information about application fetching patterns. It uses this information to request additional blocks of code and data from the streaming application server that it expects will be needed soon.

[0024] The server also initiates prefetching of application code and data by examining the patterns of requests made by the client and selectively returns to the client additional blocks that the client did not request but is likely to need soon.

[0025] A client-to-client communication mechanism allows local application customization to travel from one client machine to another without involving server communication.

[0026] A local copy-on-write file system allows some applications to write configuration or initialization files where they want to without rewriting the application, and without disturbing the local customization of other clients.

[0027] The invention disallows modifications to certain application files to prevent virus infections and reduce the chance of accidental application corruption. The system does not allow any data to be written to files that are marked as not modifiable. Attempts to mark the file as modifiable will not succeed.

[0028] The invention maintains checksums of application code and data and repairs damaged or deleted files by retrieving another copy from the application streaming server.

[0029] Applications are patched or upgraded via a change in the root directory for that application. The client can be notified of application upgrades by the streaming application server. The upgrades can be marked as mandatory, in which case the client will force the application to be upgraded.

[0030] The streaming application server broadcasts an application program's code and data and any client that is interested in that particular application program stores the broadcasted code and data for later use.

[0031] Other aspects and advantages of the invention will become apparent from the following detailed description in combination with the accompanying drawings, illustrating, by way of example, the principles of the invention.

BRIEF DESCRIPTION OF THE DRAWINGS

[0032] **FIG. 1** is a block schematic diagram of a preferred embodiment of the invention showing components on the server that deal with users subscribing to and running applications according to the invention;

[0033] **FIG. 2** is a block schematic diagram of a preferred embodiment of the invention showing the client components supporting application delivery and execution according to the invention;

[0034] **FIG. 3** is a block schematic diagram of a preferred embodiment of the invention showing the components needed to install applications on the client according to the invention;

[0035] **FIG. 4** is a block schematic diagram of the Builder that takes an existing application and extracts the Application File Pages for that application according to the invention;

[0036] **FIG. 5*a*** is a block schematic diagram illustrating how the Client Network Spoofer is used to handle mapping TCP interfaces to HTTP interfaces according to the invention;

[0037] **FIG. 5*b*** is a block schematic diagram illustrating how the Client Network Spoofer is used to handle mapping TCP interfaces to HTTP interfaces according to the invention;

[0038] **FIG. 6*a*** is a block schematic diagram showing several different components of the client software according to the invention;

[0039] **FIG. 6*b*** is a block schematic diagram showing the use of volatile and non-volatile storage of code and data in the client and server according to the invention;

[0040] **FIG. 7*a*** is a block schematic diagram showing one of two ways in which data may be compressed while in transit between the server and client according to the invention;

[0041] **FIG. 7*b*** is a block schematic diagram showing the other way in which data may b e compressed while in transit between the server and client according to the invention;

[0042] **FIG. 8** is a block schematic diagram showing an organization of the streaming client software according to the invention;

[0043] **FIG. 9** is a block schematic diagram showing an alternative organization of the streaming client software according to the invention;

[0044] **FIG. 10** is a block schematic diagram showing the application streaming software consisting of a streaming block driver according to the invention;

[0045] **FIG. 11** is a block schematic diagram showing the application streaming software has been divided into a disk driver and a user mode client according to the invention;

[0046] **FIG. 12** is a block schematic diagram showing the unencrypted and encrypted client caches according to the invention;

[0047] **FIG. 13** is a block schematic diagram showing an application generating a sequence of code or data requests to the operating system according to the invention;

[0048] **FIG. 14** is a block schematic diagram showing server-based prefetching according to the invention;

[0049] **FIG. 15** is a block schematic diagram showing a client-to-client communication mechanism that allows local application customization to travel from one client machine to another without involving server communication according to the invention;

[0050] **FIG. 16** is a block schematic diagram showing a client cache with extensions for supporting local file customization according to the invention;

[0051] **FIG. 17** is a block schematic diagram showing aspects of a preferred embodiment of the invention related to load balancing and hardware fail over according to the invention;

[0052] **FIG. 18** is a block schematic diagram showing the benefits to the use of compression in the streaming of Application File Pages according to the invention;

[0053] **FIG. 19** is a block schematic diagram showing pre-compression of Application File Pages according to the invention;

[0054] **FIG. 20** is a block schematic diagram showing multi-page compression of Application File Pages according to the invention;

[0055] **FIG. 21** is a block schematic diagram showing profile-based prefetching according to the invention;

[0056] **FIG. 22** is a block schematic diagram showing the use of tokens and a License Server according to the invention;

[0057] **FIG. 23** is a block schematic diagram showing a flowchart for the Builder Install Monitor according to the invention;

[0058] **FIG. 24** is a block schematic diagram showing a flowchart for the Builder Application Profiler according to the invention;

[0059] **FIG. 25** is a block schematic diagram showing a flowchart for the Builder SAS Packager according to the invention;

[0060] **FIG. 26***a* is a block schematic diagram showing versioning support according to the invention;

[0061] **FIG. 26***b* is a block schematic diagram showing versioning support according to the invention;

[0062] **FIG. 27** is a block schematic diagram showing a data flow diagram for the Streamed Application Set Builder according to the invention;

[0063] **FIG. 28** is a block schematic diagram showing the Streamed Application Set format according to the invention;

[0064] **FIG. 29** is a block schematic diagram showing an SAS client using a device driver paradigm according to the invention;

[0065] **FIG. 30** is a block schematic diagram showing an SAS client using a file system paradigm according to the invention;

[0066] **FIG. 31***a* through **31***h* is a schematic diagram showing various components of the AppinstallBlock format according to the invention;

[0067] **FIG. 32** is a block schematic diagram showing the Application Install Block lifecycle according to the invention;

[0068] **FIG. 33** is a block schematic diagram showing peer caching according to the invention;

[0069] **FIG. 34** is a block schematic diagram showing proxy caching according to the invention;

[0070] **FIG. 35** is a block schematic diagram showing multicast within a LAN and a packet protocol according to the invention;

[0071] **FIG. 36** is a block schematic diagram showing concurrent requests for application server pages, for the case when the page is first requested through the proxy according to the invention;

[0072] **FIG. 37** is a block schematic diagram showing concurrent requests for application server pages, for the case when the page is first requested through the peer caching according to the invention;

[0073] **FIG. 38** is a block schematic diagram showing concurrent requests for application server pages, for the case when the page is received only through peer caching according to the invention;

[0074] **FIG. 39** is a block schematic diagram showing a client-server system using peer and proxy caching according to the invention;

[0075] **FIG. 40** is a block schematic diagram showing a preferred embodiment of the invention preventing piracy of remotely served, locally executed applications according to the invention;

[0076] **FIG. 41** is a block schematic diagram showing the filtering of accesses to remote application files according to the invention;

[0077] **FIG. 42** is a block schematic diagram showing the filtering of accesses to remote files based on process code location according to the invention;

[0078] **FIG. 43** is a block schematic diagram showing the filtering of accesses to remote files based on targeted file section according to the invention;

[0079] **FIG. 44** is a block schematic diagram showing the filtering of accesses to remote files based on surmised purpose according to the invention; and

[0080] **FIG. 45** is a block schematic diagram showing the filtering of accesses to remote files based on past access history according to the invention.

DETAILED DESCRIPTION OF THE INVENTION

[0081] The invention is embodied in a client-side performance optimization system for streamed applications. A system according to the invention enables a client system to efficiently stream and execute application programs that are remotely served from a server. In addition, the invention provides a system that easily integrates into the client system's operating system.

[0082] The invention provides a highly efficient and secure application delivery system in conjunction with the adaptively optimized execution of applications across a network such as the Internet, a corporate intranet, or a wide area network. This is done in such a way that existing applications do not need to be recompiled or recoded. Furthermore, the invention is a highly scalable, load-balancing, and fault-tolerant system that provides anti-piracy protection of the streamed applications.

[0083] When using the invention, an end-user requests applications that are resident on remote systems to be launched and run on the end-user's local system. The end-user's local system is called the client or client system, e.g., a desktop, laptop, palmtop, or information appliance. A remote system is a called a server or server system and is located within a collection of one or more servers called a server cluster.

[0084] From the point of view of the client system, the application appears to be installed locally on the client even though it was initially installed on a different computer system. The applications execute locally on the client system and not on the server system. To achieve this result, the application is converted into a form suitable for streaming over the network. The streaming-enabled form of an application is called the Streamed Application Set (SAS) and the conversion process is termed the SAS Builder. The conversion of an application into its SAS form typically takes place on a system different from either an end-user client system or an Application Service Provider Server Cluster. This system is called the SAS Conversion System or, simply, the conversion system.

[0085] Components of the invention are installed on the client system to support activities such as the installation, invocation, and execution of a SAS-based application. Other components of the invention are installed on the server system to support activities such as the verification of end user application subscription and license data and the transfer and execution of a SAS-based application on the client system. Some of the client and some of the server components run in the kernel-mode while other components run in the usual user-mode.

[0086] The term Application Service Provider (ASP) refers to an entity that uses the server components on one or more server systems, i.e., an ASP Server Cluster, to deliver applications to end-user client systems. Such an entity could be, for example, a software manufacturer, an e-commerce vendor that rents or leases software, or a service department within a company. The invention enables an ASP to deliver applications across a network, in a highly efficient and secure way; the applications are adaptively optimized for execution on an end-user's client system.

[0087] A number of techniques are employed to increase the overall performance of the delivery of an application and its subsequent execution by minimizing the effect of network latency and bandwidth. Among the techniques employed are: the SAS Builder identifies sequences of frequently accessed application pages and uses this information when generating a SAS; individual SAS pages and sequences of SAS pages are compressed and cached in an in-memory cache on the server system; various aspects of the applications are monitored during their actual use on a client and the resulting profiling data is used by the client to pre-fetch (pull) and by the server to send (push) additional pages which have a high likelihood of being used prior to their actual use; and SAS pages are cached locally on a client for their immediate use when an application is invoked.

[0088] Aggregate profile data for an application, obtained by combining the profile data from all the end-user client systems running the application, is used to increase the system performance as well. A number of additional caching techniques that improve both system scalability and performance are also employed. The above techniques are collectively referred to as collaborative caching.

[0089] In an embodiment of the invention, the SAS Builder consists of three phases: installation monitoring, execution profiling, and application stream packaging. In the final SAS Builder phase, the Application Stream Packager takes the information gathered by the Application Install Monitor and the Application Execution Profiler and creates the SAS form of the application, which consists of a Stream Enabled Application Pages File and a Stream Enabled Application Install Block.

[0090] The Stream Enabled Application Install Block is used to install a SAS-based application on a client system while selected portions of the Stream Enabled Application Pages File are streamed to a client to be run on the client system. The Stream Enabled Application Install Block is the first set of data to be streamed from the server to the client and contains, among other things, the information needed by the client system to prepare for the streaming and execution of the particular application. Individual and aggregate client dynamic profile data is merged into the existing Stream Enabled Application Install Block on the server to optimize subsequent streaming of the application.

[0091] The invention employs a Client Streaming File System that is used to manage specific application-related file accesses during the execution of an application. For example, there are certain shared library files, e.g., "foo.dll", that need to be installed on the local file system, e.g., "c:\winnt\system32\foo.dll", for the application to execute. Such file names get added to a "spoof database". For the previous example, the spoof database would contain an entry saying that "c:\winnt\system32\foo.dll" is mapped to "z:\word\winnt\system32\foo.dll" where "z:" implies that this file is accessed by the Client Streaming File System. The Client Spoofer will then redirect all accesses to "c:\winnt\system32\foo.dll" to "z:\word\winnt\system32\foo.dll". In this manner, the client system gets the effect of the file being on the local machine whereas in reality the file is streamed from the server. Several different classes of files can be treated in this way, e.g., specific application registry entries and application-based networking calls when such calls cross a firewall.

[0092] Lastly, the invention incorporates a number of software anti-piracy techniques directed at combating the piracy of applications of the type described herein that are delivered to the end-user over a network for execution on a client system. Among the anti-piracy techniques included are: client-side fine-grained filtering of file accesses directed at remotely served files; filtering of file accesses based on where the code for the process that originated the request is stored; identification of crucial portions of application files and filtering file access depending on the portions of the application targeted; filtering of file accesses based on the surmised purpose of the file access, as determined by examining the program stack or flags associated with the request; and filtering of file accesses based on the surmised purpose of the file access, as determined by examining a history of previous file accesses by the same process.

[0093] As mentioned above, the invention provides server and client technology for streaming application delivery and execution. The invention includes secure license-based streaming delivery of applications over Internet/extranets/intranets utilizing client-based execution with client caching and server-based file accesses by page.

[0094] 1. The invention provides many advantages over the present approaches, including:

[0095] Secure license-based streaming delivery over Internet/extranets/intranets:

[0096] reduces IT costs over client installation;

[0097] supports rental model of app delivery, which opens new markets and increases user convenience over purchase and client installation; and

[0098] enhances the opportunities to prevent software piracy over purchase and client installation.

[0099] Client-based execution with client caching:

[0100] increases typical application performance over server-based execution;

[0101] reduces network latency and bandwidth usage over non-cached client execution; and

[0102] allows existing applications to be run w/o rewrite/recompile/rebuild unlike other explicitly-distributed client/server application delivery approaches.

[0103] Server-based file accesses:

[0104] improve server-scaling over server-based execution;

[0105] allow transparent failover to another server whereas server-based execution does not;

[0106] make server load balancing easier than it is with server-based execution; and

[0107] allow increased flexibility in server platform selection over server-based execution.

[0108] Server-based file accesses by page:

[0109] reduce network latency over complete file downloads;

[0110] reduce network bandwidth overhead over complete file downloads; and

[0111] reduce client cache footprint over complete file downloads.

[0112] 2. Features of the Invention

[0113] A) Server Components Supporting Application Delivery and Execution.

[0114] i) referring to **FIG. 1**, the server components include:

[0115] a. Client/server network interface **110** that is common to the client **113** and the server. This is the communication mechanism through which the client and the server communicate.

[0116] b. The Subscription Server **105**—This is the server the client **113** connects to for subscribing and unsubscribing applications. This server then adds/deletes the subscription information to the Subscription Database **101** and also updates the License Database **102** with the information stating that the client **113** can/cannot run the subscribed information under the agreed upon licensing terms. This communication between the client **113** and the Subscription Server **105** happens over SSL that is an industry standard protocol for secure communication. The Subscription Server **105** is also contacted for updating any existing subscription information that is in the Subscription Database **101**.

[0117] c. The License Server **106**—This is the server the client **113** connects to for getting a license to run an application after the client has subscribed to the application. This server validates the user and his subscriptions by consulting the License Database **102**. If the client **113** does have a valid license, the License Server **106** sends an "Access token" to the client that is encrypted using an "encryption key" that the License Database **102** obtains from the Server Config Database **103**. The "Access token" contains information like the Application ID and an expiration time. Along with the "Access token," the License Server **106** also sends a list of least loaded application servers that it obtains from the Server Config Database **103** and also the expiration time that was encoded in the "Access token". The client **113** uses this expiration time to know when to ask for a new token. This communication between the client **113** and the License Server **106** happens over SSL.

[0118] d. The Application Server **107**—Once the client **113** obtains an "Access token" to run an application, it connects to the Application Server **107** and presents to it the "Access token" along with the request for the application bits. Note that the "Access token" is opaque to the client **113** since it does not have the key to decrypt it. The Application Server **107** validates the "Access token" by decrypting it using a "decryption key" obtained from the Server Config Database **103** and checking the content against a predefined value like for example the Application ID and also by making sure that the expiration time in the "Access token" has not elapsed. It then serves the appropriate bits to the client **113** to enable it to run the application. The encryption and decryption keys could be something like a private key/public key pair or a symmetric key or any other means of providing security. Note that the keys are uniform across all the servers within an ASP.

[0119] e. The Monitor Server **108**—It monitors the load in terms of percent of CPU utilization on the Application Servers **107** and the License Servers **106** on a periodic basis (for example—every minute) and adds that information to the Server Config Database **103**.

[0120] f. The Profile Server **109**—It receives profile information from the clients periodically. It adds this information to the Profile Database **104**. The Profile Server **109** based on the profile information from different clients updates the App Prefetch Page List section of the Stream A p p Install Blocks **112**.

[0121] ii) The data structures supporting the above server components include:

[0122] a. Subscription Database **101**—This is the database that stores the user information in terms of usename, list of apps subscribed, password, billing information, address, group, admin. The username is the primary key. The admin field identifies if this user has admin privileges for the group he belongs to.

[0123] b. License Database **102**—This is the database that stores licensing information, i.e., which user can run what application and under which license. This database also keeps track of usage information, i.e., which user has used which application for how long and how many times. The information looks like:

[0124] Username, application, time of usage, number of times run

[0125] Username, application, licensing policy

[0126] Username, application, is app running, no of instances, time of start The username is the primary key. The licensing policy could be something simple like expiry date or something more complicated like number of instances simultaneously allowed within a group etc.

[0127] c. Server Config Database **103**—This database stores information about which server can run which application, what is the load on all the servers, what is the encryption "key" to be used by the servers and all other information that is needed by the servers. The information looks like:

[0128] Server IP address, App/Slim server, application list, current load

[0129] Encryption key, Decryption key

[0130] The Server IP address is the primary key for the first table. The keys are common across all servers.

[0131] d. Profile Database **104**—This database stores the profile information received by the profile server from the clients periodically. The information looks like:

[0132] Application ID, File ID, Block ID number of hits

[0133] The Application ID is the primary key.

[0134] e. Application File Pages **111**—This is the one of the outputs of the "builder" as explained below and is put on the Application Server **107** so that it can serve the appropriate bits to the client.

[0135] f. Stream App Install Blocks **112**—This is the other output of the "builder" and contains the information for successfully installing applications on the client for streaming applications.

[0136] B) Client Components Supporting Application Delivery & Execution

[0137] i) With respect to **FIGS. 1 and 2**, these client components include:

[0138] a. Client/Server Network interface **202**—This is the same interface as explained above.

[0139] b. Client License Manager **205**—This component requests licenses ("Access tokens") from the License Server **106** when the client wants to run applications. The License Server **106** sends an "Access token" to the client that can be used to run the applications b y presenting it to the Application Server **107**. Along with the token, the License Server **106** also sends the expiry time of the token. The Client License Manager **205** renews the token just before the expiry period so that the client can continue running the application. When the application is complete, the Client License Manager **205** releases the token by sending a message to the License Server **106**. In addition, when a user has subscribed to an application, the Client License Manager **205** first checks to make sure that the application is installed on the machine the user is trying to run the application from and if not requests for the application installation. It does this using a list of Installed Apps that it maintains.

[0140] c. Client Cache Manager **207**—This component caches the application bits received from the Application Server **107** so that next time a request is made to the same bits, the request can be served by the cache instead of having to go to the Application Server **107**. The Client Cache Manager **207** has a limited amount of space on the disk of the client machine that it uses for the cache. When the space is fully occupied, the Client Cache Manager **207** uses a policy to replace existing portions of the cache. This policy can be something like LRU, FIFO, random etc. The Client Cache Manager **207** is responsible for getting the application bits requested by the Client Streaming File System **212**. If it does not have the bits cached, it gets them from the Application Server **107** through the network interface. However it also need to get the "Access token" from the Client License Manager **205** that it needs to send along with the request for the application bits. The Client Cache Manager **207** also updates the Prefetch History Info **209** with the requests it receives from the Client Streaming File System **212**.

[0141] d. Client Streaming File System **212**—This component serves all file system requests made by the application running on the client. The application makes calls like "read", "write" etc. to files that need to be streamed. These requests lead to page faults in the operating system and the page faults are handled by the Client Streaming File System **212** that in turn asks the Client Cache Manager **207** for the appropriate bits. The Client Cache Manager **207** will send those bits from the cache if they exist there or forward the request to the Application Server **107** through the network interface to get the appropriate bits.

[**0142**] e. Client Prefetcher **208**—This component monitors the requests made by the client to the Application Server **107** and uses heuristics to make additional requests to the Application Server **107** so that the bits can be obtained from the Application Server **107** before the client machine makes the request for them. This is mainly to hide the latency between the client and the Application Server **107**. The history information of the requests is stored in the Prefetch History Info file **209**.

[**0143**] f. Client Profiler **203**—At specific time intervals, the client profiler sends the profile information, which is the Prefetch History Info to the prefetch server at the ASP that can then update the App Prefetch Page Lists for the different applications accordingly.

[**0144**] g. Client File Spoofer **211**—Certain files on the client need to b e installed at specific locations on the client system. To be able to stream these files from the Application Server **107**, the Client Spoofer **211** intercepts all requests to these files made by a running application and redirects them to the Client Streaming File System **212** so that the bits can be streamed from the Application Server **107**.

[**0145**] h. Client Registry Spoofer **213**—Similar to files, certain registry entries need to be different when the application being streamed is running and since it is undesirable to overwrite the existing registry value, the read of the registry value is redirected to the Client Registry Spoofer **215** which returns the right value. However, this is optional as it is very likely that overwriting the existing registry value will make the system work just fine.

[**0146**] i. Client Network Spoofer **213**—Certain applications make networking calls through a protocol like TCP. To make these applications work across firewalls, these networking calls need to be redirected to the Client Network Spoofer **213** which can tunnel these requests through a protocol like HTTP that works through firewalls.

[**0147**] ii) The data structures needed to support the above client components include:

[**0148**] a. File Spoof Database **210**—The list of files the requests to which need to be redirected to the Client Streaming File System **212**. This information looks like (The source file name is the primary key)

[**0149**] Source File Name, Target File Name

[**0150**] b. Registry Spoof Database **216**—List of registry entries and their corresponding values that need to be spoofed. Each entry looks like:

[**0151**] Registry entry, new value

[**0152**] c. Network Spoof Database **214**—Like of IP addresses, the networking connections to which need to be redirected to the Client Network Spoofer **213**. Each entry looks like (IP address is the primary key):

[**0153**] IP address, Port number, new IP address, new Port number

[**0154**] d. Client Stream Cache **206**—The on-disk cache that persistently stores application bits.

[**0155**] e. Known ASPs and Installed Apps **204**—The list of ASP servers (Application, License and Subscription) and also the list of applications that are installed on the client.

[**0156**] f. Prefetch History Info **209**—The history of the requests made to the cache. This consists of which blocks were requested from which file for which application and how many times each block was requested. It also consists of predecessor-successor information indicating which block got requested after a particular block was requested.

[**0157**] C) Client Application Installation

[**0158**] Referring to **FIG. 3**, the client application installation components include:

[**0159**] i) Client License Manager **303**—This is the same component explained above.

[**0160**] ii) Client Application Installer **305**—This component is invoked when the application needs to be installed. The Client Application Installer **305** sends a specific request to the Application Server **107** for getting the Stream App Install Block **301** for the particular application that needs to be installed. The Stream App Install Block **301** consists of the App Prefetch Page List **306**, Spoof Database **308**, **309**, **310**, and App Registry Info **307**. The Client Application Installer **305** then updates the various Spoof Databases **308**, **309**, **310** and the Registry **307** with this information. It also asks the Client Prefetcher **208** to start fetching pages in the App Prefetch Page List **306** from the Application Server **107**. These are the pages that are known to be needed by a majority of the users when they run this application.

[**0161**] D) Application Stream Builder Input/Output

[**0162**] With respect to **FIG. 4**, the Builder components include the following:

[**0163**] i) Application Install Monitor **403**—This component monitors the installation of an application **401** and figures out all the files that have been created during installation **402**, registry entries that were created and all the other changes made to the system during installation.

[**0164**] ii) Application Profiler **407**—After the application is installed, it is executed using a sample script. The Application Profiler **407** monitors the application execution **408** and figures out the application pages that got referenced during the execution.

[**0165**] iii) App Stream Packager **404**—The App Stream Packager **404** takes the information gathered by the Application Profiler **407** and the Application Install Monitor **403** and forms the Application File Pages **406** and the Stream App Install Block **405** from that information.

[0166] E) Network Spoofing for client-server applications:

[0167] Referring to **FIGS. 1, 4, 5***a*, **5***b*, and **6***a*, the component that does the Network Spoofing is the TCP to HTTP converter **503, 507**. The basic idea is to take TCP packets and tunnel them through HTTP on one side and do exactly the opposite on the other. As far as the client **501** and the server **502** are concerned the communication is TCP and so existing applications that run with that assumption work unmodified. This is explained in more detail below.

[0168] On the client side, the user launches an application that resides on the Client Streaming File System. That application may be started in the same ways that applications on other client file systems may be started, e.g., opening a data file associated with the application or selecting the application from the Start/Programs menu in a Windows system. From the point of view of the client's operating system and from the point of view of the application itself, that application is located locally on the client.

[0169] Whenever a page fault occurs on behalf of any application file residing on the Client Streaming File System **604**, that file system requests the page from the Client Cache Manager **606**. The Client Cache Manager **606**, after ensuring via interaction with the Client License Manager **608** that the user's client system holds a license to run the application at the current time, checks the Client Stream Cache **611** and satisfies the page fault from that cache, if possible. If the page is not currently in the Client Stream Cache **611**, the Client Cache Manager **606** makes a request to the Client/Server Network Interface **505, 609** to obtain that page from the Application File Pages stored on an Application Server **506**.

[0170] The Client Prefetcher **606** tracks all page requests passed to the Client Cache Manager **606**. Based on the pattern of those requests and on program locality or program history, the Client Prefetcher **606** asks the Client Cache Manager **606** to send additional requests to the Client/Server Network Interface **505, 609** to obtain other pages from the Application File Pages stored on the Application Server **506**.

[0171] Files located on the Client Streaming File System **604** are typically identified b y a particular prefix (like drive letter or pathname). However, some files whose names would normally imply that they reside locally are mapped to the Client Streaming File System **604**, in order to lower the invention's impact on the user's local configuration. For instance, there are certain shared library files (dil's) that need to be installed on the local file system (c:\winnt\system32\foo.dll). It is undesirable to add that file on the user's system. The file name gets added to a "spoof database" which contains an entry saying that c:\winnt\system32\foo.dll is mapped to z:\word\winnt\system32\foo.dll where z: implies that it is the Client Streaming File System. The Client Spoofer **603** will then redirect all accesses to c:\winnt\system32\foo.dll to z:\word\winnt\system32\foo.dll. In this manner the client system gets the effect of the file being on the local machine whereas in reality the file is streamed from the server.

[0172] In a similar fashion the Client Spoofer **603** may also be used to handle mapping TCP interfaces to HTTP interfaces. There are certain client-server applications (like ERP/CRM applications) that have a component running on a client and another component running on a database server, Web server etc. These components talk to each other through TCP connections. The client application will make TCP connections to the appropriate server (for this example, a database server) when the client piece of this application is being streamed on a user's machine.

[0173] The database server could be resident behind a firewall and the only way for the client and the server to communicate is through a protocol like HTTP that can pass through firewalls. To enable the client to communicate with the database server, the client's TCP requests need to be converted to HTTP and sent to the database server. Those requests can be converted back to TCP so that the database server can appropriately process the requests just before the requests reach the database server. The Client Spoofer's **603** responsibility in this case is to trap all TCP requests going to the database server and convert it into HTTP requests and take all HTTP requests coming from the database server and convert them into TCP packets. Note that the TCP to HTTP converters **505, 507** convert TCP traffic to HTTP and vice versa by embedding TCP packets within the HTTP protocol and by extracting the TCP packets from the HTTP traffic. This is called tunneling.

[0174] When the Client License Manager **608** is asked about a client's status with respect to holding a license for a particular application and the license is not already being held, the Client License Manager **608** contacts the License Server **106** via the Client/Server Network Interface **609** and asks that the client machine be given the license. The License Server **106** checks the Subscription **101** and License **102** Databases and, if the user has the right to hold the license at the current time, it sends back an Access Token, which represents the right to use the license. This Access Token is renewed by the client on a periodic basis.

[0175] The user sets up and updates his information in the Subscription **101** and License **102** Databases via interacting with the Subscription Server **105**. Whenever a user changes his subscription information, the Subscription Server **105** signals the user's client system since the client's Known ASPs and Installed Apps information potentially needs updating. The client system also checks the Subscription **101** and License **102** Databases whenever the user logs into any of his client systems set up for Streaming Application Delivery and Execution. If the user's subscription list in the Subscription **101** and License **102** Databases list applications that have not been installed on the user's client system, the user is given the opportunity to choose to install those applications.

[0176] Whenever the user chooses to install an application, the Client License Manager **608** passes the request to the Client Application Installer **607** along with the name of the Stream App Install Block to be obtained from the Application Server **107**. The Client Application Installer **607** opens and reads that file (which engages the Client Streaming File System) and updates the Client system appropriately, including setting up the spoof database, downloading certain needed non-application-specific files, modifying the registry file, and optionally providing a list of applications pages to be prefetched to warm up the Client Stream Cache **611** with respect to the application.

[0177] The Application Stream Builder creates the Stream App Install Block **405** used to set up a client system for Streaming Application Delivery and Execution and it also creates the set of Application File Pages **406** sent to satisfy client requests by the Application Server **107**. The process that creates this information is offline and involves three components. The Application Install Monitor **403** watches a normal installation of the application and records various information including registry entries, required system configuration, file placement, and user options. The Application Profiler **407** watches a normal execution of the application and records referenced pages, which may be requested to pre-warm the client's cache on behalf of this application. The Application Stream Packager **404** takes information from the other two Builder components, plus some information it compiles with respect to the layout of the installed application and forms the App Install Block **405** and the set of Application File Pages **406**.

[0178] Server fail-over and server quality of service problems are handled by the client via observation and information provided by the server components. An ASP's Subscription Server provides a list of License Servers associated with that ASP to the client, when the user initiates/modifies his account or when the client software explicitly requests a new list. A License Server provides a list of Application Servers associated with an application to the client, whenever it sends the client an Access Token for the application.

[0179] Should the client observe apparent non-response or slow response from an Application Server, it switches to another Application Server in its list for the application in question. If none of the Application Servers in its list respond adequately, the client requests a new set for the application from a License Server. The strategy is similar in the case in which the client observes apparent non-response or slow response from a License Server; the client switches to another License Server in its list for the ASP in question. If none of the License Servers in its list responds adequately, the client requests a new set of License Servers from the ASP.

[0180] Server load balancing is handled by the server components in cooperation with the client. A server monitor component tracks the overall health and responsiveness of all servers. When a server is composing one of the server lists mentioned in the previous paragraph, it selects a set that is alive and relatively more lightly used than others. Client cooperation is marked by the client using the server lists provided by the servers in the expected way, and not unilaterally doing something unexpected, like continuing to use a server which does not appear in the most recent list provided.

[0181] Security issues associated with the server client relationship are considered in the invention. To ensure that the communication between servers and clients is private and that the servers in question are authorized via appropriate certification, an SSL layer is used. To ensure that the clients are licensed to use a requested application, user credentials (username+password) are presented to a License Server, which validates the user and his licensing status with respect to the application in question and issues an Access Token, and that Access Token is in turn presented to an Application Server, which verifies that the Token's validity before delivering the requested page. Protecting the application in question from piracy on the client's system is discussed in another section, below.

CLIENT-SIDE PERFORMANCE OPTIMIZATION

[0182] This section focuses on client-specific portions of the invention. The invention may be applied to any operating system that provides a file system interface or block driver interface. A preferred embodiment of the invention is Windows 2000 compliant.

[0183] With respect to **FIG. 6***a*, several different components of the client software are shown. Some components will typically run as part of the operating system kernel, and other portions will run in user mode.

[0184] The basis of the client side of the streamed application delivery and execution system is a mechanism for making applications appear as though they were installed on the client computer system without actually installing them.

[0185] Installed applications are stored in the file system of the client system as files organized in directories. In the state of the art, there are two types of file systems: local and network. Local file systems are stored entirely on media (disks) physically resident in the client machine. Network file systems are stored on a machine physically separate from the client, and all requests for data are satisfied by getting the data from the server. Network file systems are typically slower than local file systems. A traditional approach to use the better performance of a local file system is to install important applications on the local file system, thereby copying the entire application to the local disk. The disadvantages of this approach are numerous. Large applications may take a significant amount of time to download, especially across slower wide area networks. Upgrading applications is also more difficult, since each client machine must individually be upgraded.

[0186] The invention eliminates these two problems by providing a new type of file system: a streaming file system. The streaming file system allows applications to be run immediately by retrieving application file contents from the server as they are needed, not as the application is installed. This removes the download cost penalty of doing local installations of the application. The streaming file system also contains performance enhancements that make it superior to running applications directly from a network file system. The streaming file system caches file system contents on the local machine. File system accesses that hit in the cache are nearly as fast as those to a local file system. The streaming file system also has sophisticated information about application file access patterns. By using this knowledge, the streaming file system can request portions of application files from the server in advance of when they will actually be needed, thus further improving the performance of applications running on the application streaming file system.

[0187] In a preferred embodiment of the invention, the application streaming file system is implemented on the client using a file system driver and a helper application running in user mode. The file system driver receives all requests from the operating system for files belonging to the application streaming file system. The requests it handles are all of the standard file system requests that every file system must handle, including (but not limited to) opening and closing files, reading and writing files, renaming files, and deleting files. Each file has a unique identifier consisting of an application number, and a file number within that application. In one embodiment of the invention, the application number is 128 bits and the file number is 32 bits, resulting in a unique file ID that is 160 bits long. The file system driver is responsible for converting path names (such as "z:\program files\foo.exe") into file IDs (this is described below). Once the file system driver has made this translation, it basically forwards the request to the user-mode program to handle.

[0188] The user-mode program is responsible for managing the cache of application file contents on the local file system and contacting the application streaming server for file contents that it cannot satisfy out of the local cache. For each file system request, such as read or open, the user-mode process will check to see if it has the requested information in the cache. If it does, it can copy the data from the cache and return it to the file system driver. If it does not, it contacts the application streaming server over the network and obtains the information it needs. To obtain the contents of the file, the user-mode process sends the file identifier for the file it is interested in reading along with an offset at which to read and the number of bytes to read. The application streaming server will send back the requested data.

[0189] The file system can be implemented using a fragmented functionality to facilitate development and debugging. All of the functionality of the user-mode component can be put into the file system driver itself without significantly changing the scope of the invention. Such an approach is believed to be preferred for a client running Windows 95 as the operating system.

[0190] Directories are specially formatted files. The file system driver reads these from the user mode process just like any other files with reads and writes. Along with a header containing information about the directory (such as how long it is), the directory contains one entry for each file that it contains. Each entry contains the name of the file and its file identifier. The file identifier is necessary so that the specified file can be opened, read, or written. Note that since directories are files, directories may recursively contain other directories. All files in an application streaming file system are eventual descendents of a special directory called the "root". The root directory is used as the starting point for parsing file names.

[0191] Given a name like "z:/foo/bar/baz", the file system driver must translate the path "z:/foo/bar/baz" into a file identifier that can be used to read the file from the application streaming service. First, the drive letter is stripped off, leaving "/foo/bar/baz". The root directory will be searched for the first part of the path, in this case "foo". If the file "foo" is found in the root directory, and the file "foo" is a directory, then "foo" will be searched for the next portion of the path, "bar". The file system driver achieves this by using the file id for "foo" (found by searching the root directory) to open the file and read its contents. The entries inside "foo" are then searched for "bar", and this process continues until the entire path is parsed, or an error occurs.

[0192] In the following examples and text, the root directory is local and private to the client. Each application that is installed will have its own special subdirectory in the root directory. This subdirectory will be the root of the application. Each application has its own root directory.

[0193] The invention's approach is much more efficient than other approaches like the standard NFS approach. In those cases, the client sends the entire path "/foo/bar/baz" to the server and the server returns the file id for that file. The next time there is a request for "/foo/bar/baz2" the entire path again needs to be sent. In the approach described here, once the request for "bar" was made, the file ids for all files within bar are sent back including the ids for "baz" and "baz2" and hence "baz2" will already be known to client. This reduces communication between the client and the server.

[0194] In addition, this structure also allows applications to be easily updated. If certain code segments need to be updated, then the code segment listing in the application root directory is simply changed and the new code segment subdirectory added. This results in the new and correct code segment subdirectory being read when it is referenced. For example if a file by the name of "/foo/bar/baz3" needs to be added, the root directory is simply changed to point to a new version of "foo" and that new version of "foo" points to a new version of "bar" which contains "baz3" in addition to the files it already contained. However the rest of the system is unchanged.

[0195] Client Features

[0196] Referring to **FIGS. 6***a* and **6***b*, a key aspect of the preferred embodiment of the invention is that application code and data are cached in the client's persistent storage **616**, **620**. This caching provides better performance for the client, as accessing code and data in the client's persistent storage **620** is typically much faster than accessing that data across a wide area network. This caching also reduces the load on the server, since the client need not retrieve code or data from the application server that it already has in its local persistent storage.

[0197] In order to run an application, its code and data must be present in the client system's volatile storage **619**. The client software maintains a cache of application code and data that normally reside in the client system's nonvolatile memory **620**. When the running application requires data that is not present in volatile storage **619**, the client streaming software **604** is asked for the necessary code or data. The client software first checks its cache **611**, **620** in nonvolatile storage for the requested code or data. If it is found there, the code or data are copied from the cache in nonvolatile storage **620** to volatile memory **619**. If the requested code or data are not found in the nonvolatile cache **611**, **620**, the client streaming software **604** will acquire the code or data from the server system via the client's network interface **621**, **622**.

[0198] Application code and data may be compressed **623**, **624** on the server to provide better client performance over slow networks. Network file systems typically do not compress the data they send, as they are optimized to operate over local area networks.

[0199] **FIGS. 7***a* & **7***b* demonstrate two ways in which data may be compressed while in transit between the server and client. With either mechanism, the client may request multiple pieces of code and data from multiple files at once. **FIG. 7A** illustrates the server **701** compressing the concatenation of A, B, C, and D **703** and sending this to the client **702**. **FIG. 7B** illustrates the server **706** separately compressing A, B, C, and D **708** and sending the concatenation of these compressed regions to the client **707**. In either case, the client **702**, **707** will decompress the blocks to retrieve the original contents A, B, C, and D **704**, **709** and these contents will be stored in the cache **705**, **710**.

[0200] The boxes marked "Compression" represent any method of making data more compact, including software algorithms and hardware. The boxes marked "Decompression" represent any method for expanding the compacted data, including software algorithms and hardware. The decompression algorithm used must correspond to the compression algorithm used.

[0201] The mechanism for streaming of application code and data may be a file system. Many network file systems exist. Some are used to provide access to applications, but such systems typically operate well over a local area network (LAN) but perform poorly over a wide area network (WAN). While this solution involves a file system driver as part of the client streaming software, it is more of an application delivery mechanism than an actual file system.

[0202] With respect to **FIG. 8**, application code and data are installed onto the file system **802**, **805**, **806**, **807** of a client machine, but they are executed from the volatile storage (main memory). This approach to streamed application delivery involves installing a special application streaming file system **803**, **804**. To the client machine, the streaming file system **803**, **804** appears to contain the installed application **801**. The application streaming file system **803** will receive all requests for code or data that are part of the application **801**. This file system **803** will satisfy requests for application code or data by retrieving it from its special cache stored in a native file system or by retrieving it directly from the streaming application server **802**. Code or data retrieved from the server **802** will be placed in the cache in case it is used again.

[0203] Referring to **FIG. 9**, an alternative organization of the streaming client software is shown. The client software is divided into the kernel-mode streaming file system driver **905** and a user-mode client **902**. Requests made to the streaming file system driver **905** are all directed to the user-mode client **902**, which handles the streams from the application streaming server **903** and sends the results back to the driver **905**. The advantage of this approach is that it is easier to develop and debug compared with the pure-kernel mode approach. The disadvantage is that the performance will be worse than that of a kernel-only approach.

[0204] As shown in **FIGS. 10 and 11**, the mechanism for streaming of application code and data may be a block driver **1004**, **1106**. This approach is an alternative to that represented by **FIGS. 8 and 9**.

[0205] With respect to **FIG. 10**, the application streaming software consists of a streaming block driver **1004**. This block driver **1004** provides the abstraction of a physical disk to a native file system **1003** already installed on the client operating system **1002**. The driver **1004** receives requests for physical block reads and writes, which it satisfies out of a cache on a standard file system **1003** that is backed by a physical disk drive **1006**, **1007**. Requests that cannot be satisfied by the cache go to the streaming application server **1005**, as before.

[0206] Referring to **FIG. 11**, the application streaming software has been divided into a disk driver **1106** and a user mode client **1102**. In a manner similar to that of **FIG. 9**, the disk driver **1106** sends all requests it gets to the user-mode client **1102**, which satisfies them out of the cache **1107**, **1108** or by going to the application streaming server **1103**.

[0207] The persistent cache may be encrypted with a key not permanently stored on the client to prevent unauthorized use or duplication of application code or data. Traditional network file systems do not protect against the unauthorized use or duplication of file system data.

[0208] With respect to **FIG. 12**, unencrypted and encrypted client caches. A, B, C, and D **1201** representing blocks of application code and data in their natural form are shown. $E_k(X)$ represents the encryption of block X with key k **1202**. Any encryption algorithm may be used. The key k is sent to the client upon application startup, and it is not stored in the application's persistent storage.

[0209] Client-initiated prefetching of application code and data helps to improve interactive application performance. Traditional network file systems have no prefetching or simple locality based prefetching.

[0210] Referring to **FIG. 13**, the application **1301** generates a sequence of code or data requests **1302** to the operating system(OS) **1303**. The OS **1303** directs these **1304** to the client application streaming software **1305**. The client software **1305** will fetch the code or data **1306** for any requests that do not hit in the cache from the server **1307**, via the network. The client software **1305** inspects these requests and consults the contents of the cache **1309** as well as historic information about application fetching patterns **1308**. It will use this information to request additional blocks of code and data that it expects will be needed soon. This mechanism is referred to as "pull prefetching."

[0211] Server-initiated prefetching of application code and data helps to improve interactive application performance. Traditional network file systems have no prefetching or simple locality based prefetching.

[0212] With respect to **FIG. 14**, the server-based prefetching is shown. As in **FIG. 13**, the client application streaming software **1405** makes requests for blocks **1407** from the application streaming server **1408**. The server **1408** examines the patterns of requests made by this client and selectively returns to the client additional blocks **1406** that the client did not request but is likely to need soon. This mechanism is referred to as "push prefetching."

[0213] A client-to-client communication mechanism allows local application customization to travel from one client machine to another without involving server communication. Some operating systems have a mechanism for copying a user's configuration and setup to another machine. However, this mechanism typically doesn't work outside of a single organization's network, and usually will copy the entire environment, even if only the settings for a single application are desired.

[0214] Referring to **FIG. 15, a** client-to-client mechanism is demonstrated. When a user wishes to run an application on a second machine, but wishes to retain customizations made previously on the first, the client software will handle this b y contacting the first machine to retrieve customized files and other customization data. Unmodified files will be retrieved as usual from the application streaming server.

[0215] Here, File 4 exists in three different versions. The server **1503** provides one version of this file **1506**, client **11501** has a second version of this file **1504**, and client **21502** has a third version **1505**. Files may be modified differently for each client.

[0216] The clients may also contain files not present on the server or on other clients. File **51507** is one such file; it exists only on client **11501**. File **61508** only exists on client **21502**.

[0217] Local Customization

[0218] A local copy-on-write file system allows some applications to write configuration or initialization files where they want to without rewriting the application, and without disturbing the local customization of other clients. Installations of applications on file servers typically do not allow the installation directories of applications to be written, so additional reconfiguration or rewrites of applications are usually necessary to allow per-user customization of some settings.

[0219] With respect to **FIG. 16**, the cache **1602** with extensions for supporting local file customization is shown. Each block of data in the cache is marked as "clean"**1604** or "dirty"**1605**. Pages marked as dirty have been customized by the client **1609**, and cannot be removed from the cache **1602** without losing client customization. Pages marked as clean may be purged from the cache **1602**, as they can be retrieved again from the server **1603**. The index **1601** indicates which pages are clean and dirty. In **FIG. 16**, clean pages are white, and dirty pages are shaded. File **11606** contains only clean pages, and thus may be entirely evicted from the cache **1602**. File **21607** contains only dirty pages, and cannot be removed at all from the cache **1602**. File **31608** contains some clean and some dirty pages **1602**. The clean pages of File **31608** may be removed from the cache **1602**, while the dirty pages must remain.

[0220] Selective Write Protection

[0221] The client streaming software disallows modifications to certain application files. This provides several benefits, such as preventing virus infections and reducing the chance of accidental application corruption. Locally installed files are typically not protected in any way other than conventional backup. Application file servers may be protected against writing by client machines, but are not typically protected against viruses running on the server itself. Most client file systems allow files to be marked as read-only, but it is typically possible to change a file from read-only to read-write. The client application streaming software will not allow any data to be written to files that are marked as not modifiable. Attempts to mark the file as writeable will not be successful.

[0222] Error Detection and Correction

[0223] The client streaming software maintains checksums of application code and data and can repair damaged or deleted files by retrieving another copy from the application streaming server. Traditional application delivery mechanisms do not make any provisions for detecting or correcting corrupted application installs. The user typically detects a corrupt application, and the only solution is to completely reinstall the application. Corrupt application files are detected by the invention automatically, and replacement code or data are invisibly retrieved by the client streaming software without user intervention.

[0224] When a block of code or data is requested by the client operating system, the client application streaming software will compute the checksum of the data block before it is returned to the operating system. If this checksum does not match that stored in the cache, the client will invalidate the cache entry and retrieve a fresh copy of the page from the server.

[0225] File Identifiers

[0226] Applications may be patched or upgraded via a change in the root directory for that application. Application files that are not affected by the patch or upgrade need not be downloaded again. Most existing file systems do not cache files locally.

[0227] Each file has a unique identifier (number). Files that are changed or added in the upgrade are given new identifiers never before used for this application. Files that are unchanged keep the same number. Directories whose contents change are also considered changes. If any file changes, this will cause its parent to change, all the way up to the root directory.

[0228] Upgrade Mechanism

[0229] When the client is informed of an upgrade, it is told of the new root directory. It uses this new root directory to search for files in the application. When retrieving an old file that hasn't changed, it will find the old file identifier, which can be used for the existing files in the cache. In this way, files that do not change can be reused from the cache without downloading them again. For a file that has changed, when the file name is parsed, the client will find a new file number. Because this file number did not exist before the upgrade, the client will not have this file in the cache, and will stream the new file contents when the file is freshly accessed. This way it always gets the newest version of files that change.

[0230] The client application streaming software can be notified of application upgrades by the streaming application server. These upgrades can be marked as mandatory, in which case the client software will force the application to b e upgraded.

[0231] The client will contact the application streaming server when it starts the application. At this time, the streaming application server can inform the client of any upgrades. If the upgrade is mandatory, the client will be informed, and it will automatically begin using the upgraded application by using the new root directory.

[0232] Multicast Technique

[0233] A broadcast or multicast medium may be used to efficiently distribute applications from one application streaming server to multiple application streaming clients. Traditional networked application delivery mechanisms usually involve installing application code and data on a central server and having client machines run the application from that server. The multicast mechanism allows a single server to broadcast or multicast the contents of an application to many machines simultaneously. The client machines will receive the application via the broadcast and save it in their local disk cache. The entire application can be distributed to a large number of client machines from a single server very efficiently.

[0234] The multicast network is any communication mechanism that has broadcast or multicast capability. Such media include television and radio broadcasts and IP multicasting on the Internet. Each client that is interested in a particular application may listen to the multicast media for code and data for that application. The code and data are stored in the cache for later use when the application is run.

[0235] These client techniques can be used to distribute data that changes rarely. Application delivery is the most appealing use for these techniques, but they could easily be adopted to distribute other types of slowly changing code and data, such as static databases.

LOAD BALANCING AND FAULT TOLERANCE FOR STREAMED APPLICATIONS

[0236] This section focuses on load balancing (and thereby scalability) and hardware fail over. Throughout this discussion reference should be made to **FIG. 17**. Load balancing and fault tolerance are addressed in the invention by using a smart client and smart server combination. A preferred embodiment of the invention that implements these features includes three types of servers (described below): app servers; SLM servers; and an ASP Web server. These are organized as follows:

[0237] 1: ASP Web server **1703**—This is the Web server that the user goes to for subscribing to applications, creating accounts etc. Compared to the other two types of servers it is characterized by: lowest traffic, fewest number of them, & least likely to go down.

[0238] 2: SLM Servers **1707**—subscription license manager servers—These keep track of which user has subscribed to what applications under what license etc. Compared to the other two types of servers it is characterized by: medium traffic, manageable number, and less likely to go down.

[0239] 3: App Servers **1710**—These are the servers to which the users go to for application pages. Compared to the other two types of servers it is characterized by: highest traffic, most number of them, most likely to go down either due to hardware failure or application re-configuration.

[0240] Server Lists

[0241] Clients **1704** subscribe and unsubscribe to applications via the ASP Web server **1703**. At that point, instead of getting a primary and a secondary server that can perform the job, the ASP Web server **1703** gives them a non-prioritized list of a large number of SLM servers **1706** that can do the job. When the application starts to run, each client contacts the SLM servers **1707**, **1708**, **1709** and receive its application server list **1705** that can serve the application in question and also receive the access tokens that can be used to validate themselves with the application servers **1710-1715**. All access tokens have an expiration time after which they need to be renewed.

[0242] Server Selection

[0243] Having gotten a server list for each type of server **1705**, **1706**, the client **1704** will decide which specific server to send its request to. In a basic implementation, a server is picked randomly from the list, which will distribute the client's load on the servers very close to evenly. An alternative preferred implementation will do as follows:

[0244] a) Clients will initially pick servers from the list randomly, but they will also keep track of the overall response time they get from each request; and

[0245] b) As each client learns about response times for each server, it can be more intelligent (rather than random) and pick the most responsive server. It is believed that the client is better suited at deciding which server is most responsive because it can keep track of the round trip response time.

[0246] Client-side Hardware Fail Over

[0247] The server selection logic provides hardware failover in the following manner:

[0248] a) If a server does not respond, i.e., times out, the client **1704** will pick another server from its list **1705**, **1706** and re-send the request. Since all the servers in the client's server list **1705**, **1706** are capable of processing the client's request, there are no questions of incompatibility.

[0249] b) If a SAS client **1704** gets a second time out, i.e., two servers are down, it re-sends the request to multiple servers from its list **1705**, **1706** in parallel. This approach staggers the timeouts and reduces the overall delay in processing a request.

[0250] c) In case of a massive hardware failure, all servers in the client's list **1705**, **1706** may be down. At this point, the client will use the interfaces to refresh its server list. This is where the three tiers of servers become important:

[0251] 1) If the client **1704** want to refresh its App server list **1705**, it will contact an SLM server **1707**, **1709** in its list of SLM servers **1706**. Again, the same random (SLM) server selection order is utilized here. Most of the time, this request will be successful and the client will get an updated list of app servers.

[0252] 2) If for some reason all of the SLM servers **1707**, **1709** in the client's list **1706** are down, it will contact the ASP Web server **1703** to refresh its SLM server list **1706**.

[0253] This 3-tiered approach significantly reduces the impact of a single point of failure—the ASP Web server **1703**, effectively making it a fail over of a fail over.

[0254] Server Load Balancing

[0255] In a preferred embodiment of the invention, a server side monitor **1702** keeps track of the overall health and response times for each server request. The Monitor performs this task for all Application and SLM servers. It posts prioritized lists of SLM servers and app servers **1701** that can serve each of the apps in a database shared by the monitor **1702** and all servers. The monitor's algorithm for prioritizing server lists is dominated by the server's response time for each client request. If any servers fail, the monitor **1702** informs the ASP **1703** and removes it from the server list **1701**. Note that the server lists **1705**, **1706** that the client **1704** maintains are subsets of lists the monitor **1702** maintains in a shared database **1701**.

[0256] Since all servers can access the shared database **1701**, they know how to 'cut' a list of servers to a client. For example, the client starts to run an SAS application or it wants to refresh its app server list: It will contact an SLM server and the S L M server will access the database **1701** and cut a list of servers that are most responsive (from the server's prospective).

[**0257**] In this scheme, the server monitor **1702** is keeping track of what it can track the best: how effectively servers are processing client requests (server's response time). It does not track the network propagation delays etc. that can significantly contribute to a client's observed response time.

[**0258**] ASP Managing Hardware Failovers

[**0259**] The foregoing approaches provide an opportunity for ASPs to better manage massive scale failures. Specifically, when an ASP **1703** realizes that massive numbers of servers are down, it can allocate additional resource on a temporary basis. The ASP **1703** can update the central database **1701** such that clients will receive only the list that the ASP **1703** knows to be up and running. This includes any temporary resources added to aid the situation. A particular advantage of this approach is that ASP **1703** doesn't need special actions, e.g., emails or phone support, to route clients over to these temporary resources; the transition happens automatically.

[**0260**] Handling Client Crashes and Client Evictions

[**0261**] To prevent the same user from running the same application from multiple machines, the SLM servers **1707, 1708, 1709** track what access tokens have been handed to what users. The SAS file system tracks the beginning and end of applications. The user's SAS client software asks for an access token from the SLM servers **1707, 1708, 1709** at the beginning of an application if it already does not have one and it releases the access token when the application ends. The SLM server makes sure that at a given point only one access token has been given to a particular user. In this manner, the user can run the application from multiple machines, but only from one at a particular time. However, if the user's machine crashes before the access token has been relinquished or if for some reason the ASP **1703** wants to evict a user, the access token granted to the user must be made invalid. To perform this, the SLM server gets the list of application servers **1705** that have been sent to the client **1704** for serving the application and sends a message to those application servers **1710, 1711, 1713, 1714** to stop serving that particular access token. This list is always maintained in the database so that every SLM server can find out what list is held by the user's machine. The application servers before servicing any access token must check with this list to ensure that the access token has not become invalid. Once the access token expires, it can be removed from this list.

SERVER-SIDE PERFORMANCE OPTIMIZATION

[**0262**] This section describes approaches that can be taken to reduce client-side latency (the time between when an application page is needed and when it is obtained) and improve Application Server scalability (a measure of the number of servers required to support a given population of clients). The former directly affects the perceived performance of an application by an end user (for application features that are not present in the user's cache), while the latter directly affects the cost of providing application streaming services to a large number of users.

[**0263**] Application Server Operation

[**0264**] The basic purpose of the Application Server is to return Application File Pages over the network as requested by a client. The Application Server holds a group of Stream Application Sets from which it obtains the Application File Pages that match a client request. The Application Server is analogous to a typical network file system (which also returns file data), except it is optimized for delivery of Application file data, i.e., code or data that belong directly to the application, produced by the software provider, as opposed to general user file data (document files and other content produced by the users themselves). The primary differences between the Application Server and a typical network file system are:

[**0265**] 1. The restriction to handle only Application file data allows the Application Server to only service read requests, with writes being disallowed or handled on the client itself in a copy-on-write manner;

[**0266**] 2. Access checks occur at the application level, that is a client is given all-or-none access to files for a given software application;

[**0267**] 3. The Application Server is designed to operate across the Internet, as opposed to typical network file systems, which are optimized to run over LANs. This brings up additional requirements of handling server failures, maximizing network bandwidth and minimizing latency, and handling security; and

[**0268**] 4. The Application Server is application-aware, unlike typical network file systems, which treat all software application files the same as all other files. This allows the Application Server to use and collect per-application access profile information along with other statistics.

[**0269**] To service a client request, the Application Server software component keeps master copies of the full Application Stream Sets on locally accessible persistent storage. In main memory, the Application Server maintains a cache of commonly accessed Application File Pages. The primary steps taken by the Application Server to service a client request are:

[**0270**] 1. Receive and decode the client request;

[**0271**] 2. Validate the client's privilege to access the requested data, e.g., by means of a Kerberos-style ticket issued by a trusted security service;

[**0272**] 3. Look up the requested data in the main memory cache, and failing that, obtain it from the master copy on disk while placing it in the cache; and

[**0273**] 4. Return the File Pages to the client over the network.

[**0274**] The techniques used to reduce latency and improve server scalability (the main performance considerations) are described below.

[**0275**] Server Optimization Features

[**0276**] Read-Only File System for Application Files—Because virtually all application files (code and data) are never written to by users, virtually the entire population of users have identical copies of the application files. Thus a system intending to deliver the application files can distribute a single, fixed image across all servers. The read-only file system presented by the Application Server represents this sharing, and eliminates the complexities of replication management, e.g., coherency, that occur with traditional network file systems. This simplification enables the Application Servers to respond to requests more quickly, enables potential caching at intervening nodes or sharing of caches across clients in a peer-to-peer fashion, and facilitates fail over, since with the read-only file system the Application File Pages as identified by the client (by a set of unique numbers) will always globally refer to the same content in all cases.

[0277] Per-page Compression—Overall latency observed by the client can be reduced under low-bandwidth conditions by compressing each Application File Page before sending it. Referring to **FIG. 18**, the benefits of the use of compression in the streaming of Application File Pages, is illustrated. The client **1801** and server **1802** timelines are shown for a typical transfer of data versus the same data sent in a compressed form. The client requests the data from the server **1803**. The server processes the request **1804** and begins sending the requested data. The timelines then diverge due to the ability to stream the compressed data **1805** faster than the uncompressed data **1806**.

[0278] With respect to **FIG. 19**, the invention's pre-compression of Application File Pages process is shown. The Builder generates the stream application sets **1901**, **1902** which are then pre-compressed by the Stream Application Set Post-Processor **1903**. The Stream Application Set Post-Processor **1903** stores the compressed application sets in the persistent storage device **1904**. Any client requests for data are serviced by the Application Server which sends the pre-compressed data to the requesting client **1905**. The reduction in size of the data transmitted over the network reduces the time to arrival (though at the cost of some processing time on the client to decompress the data). When the bandwidth is low relative to processing power, e.g., 256 kbps with a Pentium-III-600, this can reduce latency significantly.

[0279] Page-set Compression—When pages are relatively small, matching the typical virtual memory page size of 4 kB, adaptive compression algorithms cannot deliver the same compression ratios that they can for larger blocks of data, e.g., 32 kB or larger. Referring to **FIG. 20**, when a client **2001** requests multiple Application File Pages at one time **2002**, the Application Server **2006** can concatenate all the requested pages and compress the entire set at once **2004**, thereby further reducing the latency the client will experience due to the improved compression ratio. If the pages have already been compressed **2003**, then the request is fulfilled from the cache **2007** where the compressed pages are stored. The server **2006** responds to the client's request through the transfer of the compressed pages **2005**.

[0280] Post-processing of Stream Application Sets—The Application Server may want to perform some post processing of the raw Stream Application Sets in order to reduce its runtime-processing load, thereby improving its performance. One example is to pre-compress all Application File Pages contained in the Stream Application Sets, saving a great deal of otherwise repetitive processing time. Another possibility is to rearrange the format to suit the hardware and operating system features, or to reorder the pages to take advantage of access locality.

[0281] Static and Dynamic Profiling—With respect to **FIG. 21**, since the same application code is executed in conjunction with a particular Stream Application Set **2103** each time, there will be a high degree of temporal locality of referenced Application File Pages, e.g., when a certain feature is invoked, most if not all the same code and data is referenced each time to perform the operation. These access patterns can be collected into profiles **2108**, which can be shipped to the client **2106** to guide its prefetching (or to guide server-based **2105** prefetching), and they can be used to pre-package groups of Application File Pages **2103**, **2104** together and compress them offline as part of a post-processing step **2101**, **2102**, **2103**. The benefit of the latter is that a high compression ratio can be obtained to reduce client latency without the cost of runtime server processing load (though only limited groups of Application File Pages will be available, so requests which don't match the profile would get a superset of their request in terms of the pre-compressed groups of Application File Pages that are available).

[0282] Fast Server-Side Client Privilege Checks—Referring to **FIG. 22**, having to track individual user's credentials, i.e., which Applications they have privileges to access, can limit server scalability since ultimately the per-user data must be backed by a database, which can add latency to servicing of user requests and can become a central bottleneck. Instead, a separate License Server **2205** is used to offload per-user operations to grant privileges to access application data, and thereby allow the two types of servers **2205**, **2210** to scale independently. The License Server **2205** provides the client an Access Token (similar to a Kerberos ticket) that contains information about what application it represents rights for along with an expiration time. This simplifies the operations required b y the Application Server **2210** to validate a client's privileges **2212**. The Application Server **2210** needs only to decrypt the Access Token (or a digest of it) via a secret key shared **2209** with the License Server **2205** (thus verifying the Token is valid), then checking the validity of its contents, e.g., application identifier, and testing the expiration time. Clients **2212** presenting Tokens for which all checks pass are granted access. The Application Server **2210** needs not track anything about individual users or their identities, thus not requiring any database operations. To reduce the cost of privilege checks further, the Application Server **2210** can keep a list of recently used Access Tokens for which the checks passed, and if a client passes in a matching Access Token, the server need only check the expiration time, with no further decryption processing required.

[0283] Connection Management—Before data is ever transferred from a client to a server, the network connection itself takes up one and a half network round trips. This latency can adversely impact client performance if it occurs for every client request. To avoid this, clients can use a protocol such as HTTP 1.1, which uses persistent connections, i.e., connections stay open for multiple requests, reducing the effective connection overhead. Since the client-side file system has no knowledge of the request patterns, it will simply keep the connection open as long as possible. However, because traffic from clients may be bursty, the Application Server may have more open connections than the operating system can support, many of them being temporarily idle. To manage this, the Application Server can aggressively close connections that have been idle for a period of time, thereby achieving a compromise between the client's latency needs and the Application Server's resource constraints. Traditional network file systems do not manage connections in this manner, as LAN latencies are not high enough to be of concern.

[0284] Application Server Memory Usage/Load Balancing—File servers are heavily dependent on main memory for fast access to file data (orders of magnitude faster than disk accesses). Traditional file servers manage their main memory as cache of file blocks, keeping the most commonly accessed ones. With the Application Server, the problem of managing main memory efficiently becomes more complicated due to there being multiple servers providing a shared set of applications. In this case, if each server managed its memory independently, and was symmetric with the others, then each server would only keep those file blocks most common to all clients, across all applications. This would cause the most common file blocks to be in the main memory of each and every Application server, and since each server would have roughly the same contents in memory, adding more servers won't improve scalability by much, since not much more data will be present in memory for fast access. For example, if there are application A (accessed 50% of the time), application B (accessed 40% of the time), and application C (accessed 10% of the time), and application A and B together consume more memory cache than a single Application Server has, and there are ten Application Servers, then none of the Application Servers will have many blocks from C in memory, penalizing that application, and doubling the number of servers will improve C's performance only minimally. This can be improved upon by making the Application Servers asymmetric, in that a central mechanism , e.g., system administrator, assigns individual Application Servers different Application Stream Sets to provide, in accordance with popularity of the various applications. Thus, in the above example, of the ten servers, five can b e dedicated to provide A, four to B, and one to C, (any extra memory available for any application) making a much more effective use of the entire memory of the system to satisfy the actual needs of clients. This can be taken a step further b y dynamically (and automatically) changing the assignments of the servers to match client accesses over time, as groups of users come and go during different time periods and as applications are added and removed from the system. This can be accomplished by having servers summarize their access patterns, send them to a central control server, which then can reassign servers as appropriate.

CONVERSION OF CONVENTIONAL APPLICATIONS TO ENABLE STREAMED DELIVERY AND EXECUTION

[0285] The Streamed Application Set Builder is a software program. It is used to convert locally installable applications into a data set suitable for streaming over a network. The streaming-enabled data set is called the Streamed Application Set (SAS). This section describes the procedure used to convert locally installable applications into the SAS.

[0286] The application conversion procedure into the SAS consists of several phases. In the first phase, the Builder program monitors the installation process of a local installation of the desired application for conversion. The Builder monitors any changes to the system and records those changes in an intermediate data structure. After the application is installed locally, the Builder enters the second phase of the conversion. In the second phase, the Builder program invokes the installed application executable and obtains sequences of frequently accessed file blocks of this application. Both the Builder program and the client software use the sequence data to optimize the performance of the streaming process. Once the sequencing information is obtained, the Builder enters the final phase of the conversion. In this phase, the Builder gathers all data obtained from the first two phases and processes the data into the Streamed Application Set.

[0287] Detailed descriptions of the three phases of the Builder conversion process are described in the following sections. The three phases consist of installation monitoring (IM), application profiling (AP), and SAS packaging (SP). In most cases, the conversion process is general and applicable to all types of systems. In places where the conversion is OS dependent, the discussion is focused on the Microsoft Windows environment. Issues on conversion procedure for other OS environments are described in later sections.

[0288] Installation Monitoring (IM)

[0289] In the first phase of the conversion process, the Builder Installation Monitor (IM) component invokes the application installation program that installs the application locally. The IM observes all changes to the local computer during the installation. The changes may involve one or more of the following: changes to system or environment variables; and modifications, addition, or deletion of one or more files. Initial system variables, environment variables, and files are accounted for by the IM before the installation begins to give a more accurate picture of any changes that are observed. The IM records all changes to the variables and files in a data structure to be sent to the Builder's Streamed Application Packaging component. In the following paragraphs, detailed description of the Installation Monitor is described for Microsoft Windows environment.

[0290] In Microsoft Windows system, the Installation Monitor (IM) component consists of a kemel-mode driver subcomponent and a user-mode subcomponent. The kemel-mode driver is hooked into the system registry and file system function interface calls. The hook into the registry function calls allows the IM to monitor system variable changes. The hook into the file system function calls enables the IM to observe file changes.

[0291] Installation Monitor Kernel-Mode subcomponent (IM-KM)

[0292] With respect to **FIG. 23**, the IM-KM subcomponent monitors two classes of information during an application installation: system registry modifications and file modifications. Different techniques are used for each of these classes.

[0293] To monitor system registry modifications **2314**, the IM-KM component replaces all kernel-mode API calls in the System Service Table that write to the system registry with new functions defined in the IM-KM subcomponent. When an installation program calls one of the API functions to write to the registry **2315**, the IM-KM function is called instead, which logs the modification data **2317** (including registry key path, value name and value data) and then forwards the call to the actual operating system defined function **2318**. The modification data is made available to the IM-UM subcomponent through a mechanism described below.

[0294] To monitor file modifications, a filter driver is attached to the file system's driver stack. Each time an installation program modifies a file on the system, a function is called in the IM-KM subcomponent, which logs the modification data (including file path and name) and makes it available to the IM-UM using a mechanism described below.

[0295] The mechanisms used for monitoring registry modifications and file modifications will capture modifications made by any of the processes currently active on the computer system. While the installation program is running, other processes that, for example, operate the desktop and service network connections may be running and may also modify files or registry data during the installation. This data must be removed from the modification data to avoid inclusion of modifications that are not part of the application installation. The IM-KM uses process monitoring to perform this filtering.

[0296] To do process monitoring, the IM-KM installs a process notification callback function that is called each time a process is created or destroyed by the operating system. Using this callback function, the operating system sends the created process ID as well as the process ID of the creator (or parent) process. The IM-KM uses this information, along with the process ID of the IM-UM, to create a list of all of the processes created during the application installation. The IM-KM uses the following algorithm to create this list:

> **[0297]** 1. Before the installation program is launched by the IM-UM, the IM-UM passes its own process ID to the IM-KM. Since the IM-UM is launching the installation application, the IM-UM will be the ancestor (parent, grandparent, etc.) of any process (with one exception—the Installer Service described below) that modifies files or registry data as part of the application installation.
>
> **[0298]** 2. When the installation is launched and begins the creating processes, the IM-KM process monitoring logic is notified by the operating system via the process notification callback function.
>
> **[0299]** 3. If the creator (parent) process ID sent to the process notification callback function is already in the process list, the new process is included in the list.

[0300] When an application on the system modifies either the registry or files, and the IM-KM monitoring logic captures the modification data, but before making it available to the IM-UM, it first checks to see if the process that modified the registry or file is part of the process list. It is only made available to the IM-UM if it is in the process list.

[0301] It is possible that a process that is not a process ancestor of the IM-UM will make changes to the system as a proxy for the installation application. Using interprocess communication, an installation program may request than an Installer Service make changes to the machine. In order for the IM-KM to capture changes made by the Installer Service, the process monitoring logic includes a simple rule that also includes any registry or file changes that have been made by a process with the same name as the Installer Service process. On Windows 2000, for example, the Installer Service is called "msi.exe".

[0302] Installation Monitor User-Mode subcomponent (IM-UM)

[0303] The IM kernel-mode (IM-KM) driver subcomponent is controlled by the user-mode subcomponent (IM-UM). The IM-KM sends messages to the IM-KM to start **2305** and stop **2309** the monitoring process via standard I/O control messages known as IOCTLs. The message that starts the IM-KM also passes in the process ID of the IM-UM to facilitate process monitoring described in the IM-KM description.

[0304] When the installation program **2306** modifies the computer system, the IM-KM signals a named kernel event. The IM-UM listens for these events during the installation. When one of these events is signaled, the IM-KM calls the IM-KM using an IOCTL message. In response, the IM-KM packages data describing the modification and sends it to the IM-UM **2318**.

[0305] The IM-UM sorts this data and removes duplicates. Also, it parameterizes all local-system-specific registry keys, value names, and values. For example, an application will often store paths in the registry that allow it to find certain files at run-time. These path specifications must be replaced with parameters that can be recognized by the client installation software.

[0306] A user interface is provided for the IM-UM that allows an operator of the Builder to browse through the changes made to the machine and to edit the modification data before the data is packaged into an SAS.

[0307] Once the installation of an application is completed **2308**, the IM-UM forwards data structures representing the file and registry modifications to the Streamed Application Packager **2312**.

[0308] Monitoring Application Configuration

[0309] Using the techniques described above for monitoring file modifications and monitoring registry modifications, the builder can also monitor a running application that is being configured for a particular working environment. The data acquired by the IM-UM can be used to duplicate the same configuration on multiple machines, making it unnecessary for each user to configure his/her own application installation.

[0310] An example of this is a client server application for which the client will be streamed to the client computer system. Common configuration modifications can be captured by the IM and packed into the SAS. When the application is streamed to the client machine, it is already configured to attach to the server and begin operation.

[0311] Application Profiling (AP)

[0312] Referring to **FIG. 24**, in the second phase of the conversion process, the Builder's Application Profiler (AP) component invokes the application executable program that is installed during the first phase of the conversion process. Given a particular user input, the executable program file blocks are accessed in a particular sequence. The purpose of the AP is to capture the sequence data associated with some user inputs. This data is useful in several ways.

[0313] First of all, frequently used file blocks can be streamed to the client machine before other less used file blocks. A frequently used file block is cached locally on the client cache before the user starts using the streamed application for the first time. This has the effect of making the streamed application as responsive to the user as the locally installed application by hiding any long network latency and bandwidth problems.

[0314] Secondly, the frequently accessed files can be reordered in the directory to allow faster lookup of the file information. This optimization is useful for directories with large number of files. When the client machine looks up a frequently used file in a directory, it finds this file early in the directory search. In an application run with many directory queries, the performance gain is significant.

[0315] Finally, the association of a set of file blocks with a particular user input allows the client machine to request minimum amount of data needed to respond to that particular user command. The profile data association with a user command is sent from the server to the client machine in the AppInstallBlock during the 'preparation' of the client machine for streaming. When the user on a client machine invokes a particular command, the codes corresponding to this command are prefetched from the server.

[0316] The Application Profiler (AP) is not as tied to the system as the Installation Monitor (IM) but there are still some OS dependent issues. In the Windows system, the AP still has two subcomponents: kernel-mode (AP-KM) subcomponent and the user-mode (AP-UM) subcomponent. The AP-UM invokes the converting application executable. Then AP-UM starts the AP-KM **2403**, **2413** to track the sequences of file block accesses by the application **2414**. Finally when the application exits after the pre-specified amount of sequence data is gathered, the AP-UM retrieves the data from AP-KM **2406**, **2417** and forwards the data to the Streamed Application Packager **2411**.

[0317] Streamed Application Set Packaging (SP)

[0318] With respect to **FIG. 25**, in the final phase of the conversion process, the Builder's Streamed Application Set Packager (SP) component processes the data structure from IM and AP to create a data set suitable for streaming over the network. This converted data set is called the Streamed Application Set **2520** and is suitable for uploading to the Streamed Application Servers for subsequent downloading by the stream client. **FIG. 23** shows the control flow of the SP module.

[0319] Each file included in a Streamed Application Set **2520** is assigned a file number that identifies it within the SAS.

[0320] The Streamed Application Set **2520** consists of the three sets of data from the Streamed Application Server's perspective. The three types of data are the Concatenation Application File (CAF) **2519**, **2515**, the Size Offset File Table (SOFT)**2518**, **2514**, **2507**, and the Root Versioning Table (RVT) **2518**, **2514**.

[0321] The CAF **2519**, **2515** consists of all the files and directories needed to stream to the client. The CAF can be further divided into two subsets: initialization data set and the runtime data set.

[0322] The initialization data set is the first set of data to be streamed from the server to the client. This data set contains the information captured by IM and AP needed by the client to prepare the client machine for streaming this particular application. This initialization data set is also called the AppInstallBlock (AIB) **2516**, **2512**. In addition to the data captured by the IM and AP modules, the SP is also responsible for merging any new dynamic profile data gathered from the client and the server. This data is merged into the existing AppInstallBlock to optimize subsequent streaming of the application **2506**. With the list of files obtained b y the IM during application installation, the SP module separates the list of files into regular streamed files and the spoof files. The spoof files consists of those files not installed into standard application directory. This includes files installed into system directories and user specific directories. The detailed format description of the AppInstallBlock is described later.

[0323] The second part of the CAF consists of the runtime data set. This is the rest of the data that is streamed to the client once the client machine is initialized for this particular application. The runtime data consists of all the regular application files and the directories containing information about those application files. Detailed format description of the runtime data in the CAF section is described below. The SP appends every file recorded by IM into the CAF and generates all directories. Each directory contains list of file name, file number, and the metadata associated with the files in that particular directory **2517**, **2513**.

[0324] The SP is also responsible for generating the SOFT file **2518**, **2514**, **2507**. This is a table used to index into the CAF for determining the start and the end of a file. The server uses this information to quickly access the proper file within the directory for serving the proper file blocks to the client.

[0325] Finally, the SP creates the RVT file **2518**, **2514**. The Root Versioning Table contains a list of root file number and version number. This information is used to track minor application patches and upgrades. Each entry in the RVT corresponds to one patch level of the application with a corresponding new root directory. The SP generates new parent directories when any single file in that subdirectory tree is changed from the patched upgrade. The RVT is uploaded to the server and requested by the client at appropriate time for the most updated version of the application by a simple comparison of the client's Streamed Application root file number with the RVT table located on the server once the client is granted access authorization to retrieve the data.

[0326] With respect to **FIGS. 26***a* and **26***b*, the internal representation of a simple SAS before and after a new file is added to a new version of an application is shown. The original CAF **2601** has the new files **2607** appended to it **2604** by the SP. The SOFT **2602** is correspondingly updated **2605** with the appropriate table entries **2608** to index the new files **2607** the CAF **2604**. Finally, the RVT **2603** is updated **2606** to reflect the new version **2609**.

[0327] Data Flow Description

[0328] The following list describes the data that is passed from one component to another. The numbers corresponds to the numbering in the Data Flow diagram of **FIG. 27**.

[0329] Install Monitor

[0330] 1. The full pathname of the installer program is queried from the user by the Builder program and is sent to the Install Monitor.

[0331] 2. The Install Monitor (IM) user-mode sends a read request to the OS to spawn a new process for installing the application on the local machine.

[0332] 3. The OS loads the application installer program into memory and runs the application installer program. OS returns the process ID to the IM.

[0333] 4. The application program is started by the IM-UM.

[0334] 5. The application installer program sends read request to the OS to read the content of the CD.

[0335] 6. The CD media data files are read from the CD.

[0336] 7. The files are written to the appropriate location on the local hard-drive.

[0337] 8. IM kernel-mode captures all file read/write requests and all registry read/write requests by the application installer program.

[0338] 9. IM user-mode program starts the IM kernel-mode program and sends the request to start capturing all relevant file and registry data.

[0339] 10. IM kernel-mode program sends the list of all file modifications, additions, and deletions; and all registry modifications, additions, and deletions to the IM user-mode program.

[0340] 11. IM informs the SAS Builder UI that the installation monitoring has completed and displays the file and registry data in a graphical user interface.

[0341] Application Profiler

[0342] 12. Builder UI invokes Application Profiling (AP) user-mode program by querying the user for the list of application executable names to be profiled. The AP user-mode also queries the user for division of file blocks into sections corresponding to the commands invoked by the user of the application being profiled.

[0343] 13. Application Profiler user-mode invokes each application executable in succession by spawning each program in a new process. The OS loads the application executable into memory, runs the application executable, and returns the process ID to the Application Profiler.

[0344] 14. During execution, the OS on behalf of the application, sends the request to the hard-drive controller to read the appropriate file blocks into memory as needed by the application.

[0345] 15. The hard-drive controller returns all file blocks requested by the OS.

[0346] 16. Every file access to load the application file blocks into memory is monitored by the Application Profiler (AP) kernel-mode program.

[0347] 17. The AP user-mode program informs the AP kernel-mode program to start monitoring relevant file accesses.

[0348] 18. Application Profiler kernel-mode returns the file access sequence and frequency information to the user-mode program.

[0349] 19. Application Profiler returns the processed profile information. This has two sections. The first section is used to identify the frequency of files accessed. The second section is used to list the file blocks for prefetch to the client. The file blocks can be further categorized into subsections according to the commands invoked by the user of the application.

[0350] SAS Packager

[0351] 20. The Streamed Application Packager receives files and registry changes from the Builder UI. It also receives the file access frequency and a list of file blocks from the Profiler. File numbers are assigned to each file.

[0352] 21. The Streamed Application Packager reads all the file data from the hard-drive that are copied there by the application installer.

[0353] 22. The Streamed Application Packager also reads the previous version of Streamed Application Set for support of minor patch upgrades.

[0354] 23. Finally, the new Streamed Application Set data is stored back to non-volatile storage.

[0355] 24. For new profile data gathered after the SAS has been created, the packager is invoked to update the AppInstallBlock in the SAS with the new profile information.

[0356] Mapping of Data Flow to Streamed Application Set (SAS)

[0357] Step 7: Data gathered from this step consist of the registry and file modification, addition, and deletion. The data are mapped to the AppInstallBlock's File Section, Add Registry Section, and Remove Registry Section.

[0358] Step 8 & 19: File data are copied to the local hard-drive then concatenated into part of the CAF contents. Part of the data is identified as spoof or copied files and the file names and/or contents are added to the AppInstallBlock.

[0359] Step 15 & 21: Part of the data gathered by the Profiler or gathered dynamically by the client is used in the AppInstallBlock as a prefetch hint to the client. Another part of the data is used to generate a more efficient SAS Directory content by ordering the files according the usage frequency.

[0360] Step 20: If the installation program was an upgrade, SAS Packager needs previous version of the Streamed Application Set data. Appropriate data from the previous version are combined with the new data to form the new Streamed Application Set.

[0361] Format of Streamed Application Set

[0362] Referring to **FIG. 28**, the format of the Streamed Application Set consists of three sections: Root Version Table (RVT) **2802**, Size Offset File Table (SOFT) **2803**, and Concatenation Application File (CAF) **2801**. The RVT section **2802** lists all versions of the root file numbers available in a Streamed Application Set. The SOFT **2803** section consists of the pointers into the CAF **2801** section for every file in the CAF **2801**. The CAF section **2801** contains the concatenation of all the files associated with the streamed application. The CAF section **2801** is made up of regular application files, SAS directory files **2805**, AppInstallBlock **2804**, and icon files. See below for detailed information on the content of the SAS file.

[0363] OS Dependent Format

[0364] The format of the Streamed Application Set is designed to be as portable as possible across all OS platforms. At the highest level, the format of CAF, SOFT, and RVT that make up the format of the Streamed Application Set are completely portable across any OS platforms. One piece of data structure that is OS dependent is located in the initialization data set called AppInstallBlock in the CAF. This data is dependent on the type of OS due to the differences in low-level system differences among different OS. For example, Microsoft Windows contains system environment variables called the Registry. The Registry has a particular tree format not found in other operating systems like UNIX or MacOS.

[0365] Another OS dependent piece of data is located in the SAS directory files in the CAF. The directory contains file metadata information specific to Windows files. For example on the UNIX platform, there does not exist a hidden flag. This platform specific information needs to be transmitted to the client to fool the streamed application into believing that the application data is located natively on the client machine with all the associated file metadata intact. If SAS is to be used to support streaming of UNIX or MacOS applications, file metadata specific to those systems will need to be recorded in the SAS directory.

[0366] Lastly, the format of the file names itself is OS dependent. Applications running on the Windows environment inherit the old MSDOS 8.3 file name format. To support this properly, the format of the SAS Directory file in CAF requires an additional 8.3 field to store this information. This field is not needed in other operating systems like UNIX or MacOS.

[0367] Device Driver Versus File System Paradigm

[0368] Referring to **FIGS. 29 and 30**, the SAS client Prototype is implemented using the 'device driver' paradigm. One of the advantages of the device driver approach is that the caching of the sector blocks is simpler. In the device driver approach, the client cache manager **2902** only needs to track sector numbers in its cache **2903**. In comparison with the 'file system' paradigm, more complex data structure are required by the client cache manager **3002** to track a subset of a file that is cached **3003** on a client machine. This makes the device driver paradigm easier to implement.

[0369] On the other hand, there are many drawbacks to the device driver paradigm. On the Windows system, the device driver approach has a problem supporting large numbers of applications. This is due to the phantom limitation on the number of assignable drive letters available in a Windows system (26 letters);

[0370] and the fact that each application needs to be located on its own device. Note that having multiple applications on a device is possible, but then the server needs to maintain an exponential number of devices that support all possible combinations of applications. This is too costly to maintain on the server.

[0371] Another problem with the device driver approach is that the device driver operates at the disk sector level. This is a much lower level than operating at the file level in the file system approach. The device driver does not know anything about files. Thus, the device driver cannot easily interact with the file level issues.

[0372] For example, spoofing files and interacting with the OS file cache is nearly impossible with the device driver approach. Both spoofing files and interacting with the OS buffer cache are needed to get higher performance. In addition, operating at the file system level lends to optimizing the file system to better suit this approach of running applications. For instance, typical file systems do logging and make multiple disk sector requests at a time. These are not needed in this approach and are actually detrimental to the performance. When operating at the device driver level, not much can be done about that. Also, operating at the file system level helps in optimizing the protocol between the client and the server.

[0373] Implementation in the Prototype

[0374] The prototype has been implemented and tested successfully on the Windows and Linux distributed system. The prototype is implemented using the device driver paradigm as described above. The exact procedure for streaming application data is described next.

[0375] First of all, the prototype server is started on either the Windows-based or Linux-based system. The server creates a large local file mimicking large local disk images. Once the disk images are prepared, it listens to TCP/IP ports for any disk sector read or write requests.

[0376] Implementation of SAS Builder

[0377] The SAS Builder has been implemented on the Windows-based platform. A preliminary Streamed Application Set file has been created for real-world applications like Adobe Photoshop. A simple extractor program has been developed to extract the SAS data on a pristine machine without the application installed locally. Once the extractor program is run on the SAS, the application runs as if it was installed locally on that machine. This process verifies the correctness of the SAS Building process.

FORMAT OF STREAMED APPLICATION SET (SAS)

[0378] Functionality

[0379] The streamed application set (SAS), illustrated in **FIG. 28**, is a data set associated with an application suitable for streaming over the network. The SAS is generated by the

SAS Builder program. The program converts locally installable applications into the SAS. This section describes the format of the SAS.

[0380] Note: Fields greater than a single byte are stored in little-endian format. The Stream Application Set (SAS) file size is limited to 2A64 bytes. The files in the CAF section are laid out in the same order as its corresponding entries in the SOFT table.

[0381] Data Type Definitions

[0382] The format of the SAS consists of four sections: header, Root Version Table (RVT), Size Offset File Table (SOFT), and Concatenation Application File (CAF) sections.

[0383] 1. Header section

[0384] MagicNumber [4 bytes]: Magic number identifying the file content with the SAS.

[0385] ESSVersion [4 bytes]: Version number of the SAS file format.

[0386] AppID [16 bytes]: A unique application ID for this application. This field must match the AppID located in the AppInstallBlock. Window Guidgen API is used to create this identifier.

[0387] Flags [4 bytes]: Flags pertaining to SAS.

[0388] Reserved [32 bytes]: Reserved spaces for future.

[0389] RVToffset [8 bytes]: Byte offset into the start of the RVT section.

[0390] RVTsize [8 bytes]: Byte size of the RVT section.

[0391] SOFToffset [8 bytes]: Byte offset into the start of the SOFT section.

[0392] SOFTsize [8 bytes]: Byte size of the SOFT section.

[0393] CAFoffset [8 bytes]: Byte offset into the start of the CAF section.

[0394] CAFsize [8 bytes]: Byte size of the CAF section.

[0395] VendorNameIsAnsi [1 byte]: 0 if the vendor name is in Unicode format. 1 if the vendor name is in ANSI format.

[0396] VendorNameLength [4 bytes]: Byte length of the vendor name.

[0397] VendorName [X bytes]: Name of the software vendor who created this application. e.g., "Microsoft". Null-terminated.

[0398] AppBaseNameIsAnsi [1 byte]: 0 if the vendor name is in Unicode format. 1 if the vendor name is in ANSI format.

[0399] AppBaseNameLength [4 bytes]: Byte length of the application base name.

[0400] AppBaseName [X bytes]: Base name of the application. e.g., "Word 2000". Null-terminated.

[0401] MessageIsAnsi [1 byte]: 0 if the vendor name is in Unicode format. 1 if the vendor name is in ANSI format.

[0402] MessageLength [4 bytes]: Byte length of the message text.

[0403] Message [X bytes]: Message text. Null-terminated.

[0404] 2. Root Version Table (RVT) section

[0405] The Root version entries are ordered in a decreasing value according to their file numbers. The Builder generates unique file numbers within each SAS in a monotonically increasing value. So larger root file numbers imply later versions of the same application. The latest root version is located at the top of the section to allow the SAS Server easy access to the data associated with the latest root version.

[0406] NumberEntries [4 bytes]: Number of patch versions contained h this SAS. The number indicates the number of entries in the Root Version Table (RVT).

[0407] Root Version structure: (variable number of entries)

[0408] VersionNumber [4 bytes]: Version number of the root directory.

[0409] FileNumber [4 bytes]: File number of the root directory.

[0410] VersionNameIsAnsi [1 byte]: 0 if the vendor name is in Unicode format. 1 if the vendor name is in ANSI format.

[0411] VersionNameLength [4 bytes]: Byte length of the version name

[0412] VersionName [X bytes]: Application version name. e.g., "SP 1".

[0413] Metadata [32 bytes]: See SAS FS Directory for format of the metadata.

[0414] 3. Size Offset File Table (SOFT) section

[0415] The SOFT table contains information to locate specific files in the CAF section. The entries are ordered according to the file number starting from 0 to NumberFiles-1. The start of the SOFT table is aligned to eight-byte boundaries for faster access.

[0416] SOFT entry structure: (variable number of entries)

[0417] Offset [8 bytes]: Byte offset into CAF of the start of this file.

[0418] Size [8 bytes]: Byte size of this file. The file is located from address Offset to Offset+Size.

[0419] 4. Concatenation Application File (CAF) section

[0420] CAF is a concatenation of all file or directory data into a single data structure. Each piece of data can be a regular file, an AppInstallBlock, an SAS FS directory file, or an icon file.

[0421] a. Regular Files

[0422] FileData [X bytes]: Content of a regular file

[0423] b. AppInstallBlock (See AppInstallBlock section for detailed format) A simplified description of the AppInstallBlock is listed here. The exact detail of the individual fields in the AppInstallBlock are described later.

[0424] Header section [X bytes]: Header for AppInstallBlock containing information to identify this AppInstallBlock.

[0425] Files section [X bytes]: Section containing file to be copied or spoofed.

[0426] AddVariable section [X bytes]: Section containing system variables to be added.

[0427] RemoveVariable section [X bytes]: Section containing system variables to be removed.

[0428] Prefetch section [X bytes]: Section containing pointers to file blocks to be prefetched to the client.

[0429] Profile section [X bytes]: Section containing profile data.

[0430] Comment section [X bytes]: Section containing comments about AppInstallBlock.

[0431] Code section [X bytes]: Section containing application-specific code needed to prepare local machine for streaming this application

[0432] LicenseAgreement section [X bytes]: Section containing licensing agreement message.

[0433] c. SAS Directory

[0434] An SAS Directory contains information about the subdirectories and files located within this directory. This information is used to store metadata information related to the files associated with the streamed application. This data is used to fool the application into thinking that it is running locally on a machine when most of the data is resided elsewhere.

[0435] The SAS directory contains information about files in its directory. The information includes file number, names, and metadata associated with the files.

[0436] MagicNumber [4 bytes]: Magic number for SAS directory file.

[0437] ParentFileID [16+4 bytes]: AppID+FileNumber of the parent directory. AppID is set to 0 if the directory is the root.

[0438] SelfFileID [16+4 bytes]: AppID+FileNumber of this directory.

[0439] NumFiles [4 bytes]: Number of files in the directory.

[0440] Variable-Sized File Entry:

[0441] UsedFlag [1 byte]: 1 for used, 0 for unused.

[0442] ShortLen [1 byte]: Length of short file name.

[0443] LongLen [2 byte]: Length of long file name.

[0444] NameHash [4 bytes]: Hash value of the short file name for quick lookup without comparing whole string.

[0445] ShortName [24 bytes]: 8.3 short file name in Unicode. Not null-terminated.

[0446] FileID [16+4 bytes]: AppID+FileNumber of each file in this directory.

[0447] Metadata [32 bytes]: The metadata consists of file byte size (8 bytes), file creation time (8 bytes), file modified time (8 bytes), attribute flags (4 bytes), SAS flags (4 bytes). The bits of the attribute flags have the following meaning:

[0448] Bit **0**: Read-only—Set if file is read-only

[0449] Bit **1**: Hidden—Set if file is hidden from user

[0450] Bit **2**: Directory—Set if the file is an SAS Directory

[0451] Bit **3**: Archive—Set if the file is an archive

[0452] Bit **4**: Normal—Set if the file is normal

[0453] Bit **5**: System—Set if the file is a system file

[0454] Bit **6**: Temporary—Set if the file is temporary

[0455] The bits of the SAS flags have the following meaning:

[0456] Bit **0**: ForceUpgrade—Used only on root file. Set if client is forced to upgrade to this particular version if the current root version on the client is older.

[0457] Bit **1**: RequireAccessToken—Set if file require access token before client can read it.

[0458] Bit **2**: Read-only—Set if the file is read-only

[0459] LongName [X bytes]: Long filename in Unicode format with null-termination character.

[0460] d. Icon files

[0461] IconFileData [X bytes]: Content of an icon file.

FORMAT OF APPINSTALLBLOCK

[0462] Functionality

[0463] With respect to **FIGS. 31***a*-**31***h*, the AppInstallBlock is a block of code and data associated with a particular application. This AppInstallBlock contains the information needed to by the SAS client to initialize the client machine before the streamed application is used for the first time. It also contains optional profiling data for increasing the runtime performance of that streamed application.

[0464] The AppInstallBlock is created offline by the SAS Builder program. First of all, the Builder monitors the installation process of a local version of the application installation program and records changes to the system. This includes any environment variables added or removed from the system **3103**, and any files added or modified in the system directories **3102**. Files added to the application specific directory are not recorded in the AppInstallBlock to reduce the amount of time needed to send the AppInstallBlock to the SAS client. Secondly, the Builder profiles the application to obtain the list of critical pages needed to run the application initially and an initial page reference sequence of the pages accessed during a sample run of the application. The AppInstallBlock contains an optional application-specific initialization code **3107**. This code is needed when the default initialization procedure is insufficient to setup the local machine environment for that particular application.

[0465] The AppInstallBlock and the runtime data are packaged into the SAS by the Builder and then uploaded to the application server. After the SAS client is subscribed to an application and before the application is run for the first time, the AppInstallBlock is sent by the server to the client. The SAS client invokes the default initialization procedure and the optional application-specific initialization code **3107**. Together, the default and the application-specific initialization procedure process the data in the AppInstallBlock to make the machine ready for streaming that particular application.

[0466] Data type definitions

[0467] The AppInstallBlock is divided into the following sections: header section **3101**, variable section **3103**, file section **3102**, profile section **3105**, prefetch section **3104**, comment section **3106**, and code section **3107**. The header section **3101** contains general information about the AppInstallBlock. The information includes the total byte size and an index table containing size and offset into other sections. In a Windows version, the variable section consists of two registry tree structures to specify the registry entries added or removed from the OS environment. The file section **3102** is a tree structure consisting of the files copied to C drive during the application installation. The profile section **3105** contains the initial set of block reference sequences during Builder profiling of the application. The prefetch section **3104** consists of a subset of profiled blocks used by the Builder as a hint to the SAS client to prefetch initially. The comment section **3106** is used to inform the SAS client user of any relevant information about the application installation. Finally, the code section **3107** contains an optional program tailored for any application-specific installation not covered by the default streamed application installation procedure. In Windows version, the code section contains a Windows DLL. The following is a detailed description of each fields of the AppInstallBlock.

[0468] Note: Little endian format is used for all the fields spanning more than 1 byte. Also, BlockNumber specifies blocks of 4 K byte size.

[0469] Header Section

[0470] The header section **3103** contains the basic information about this AppInstallBlock. This includes the versioning information, application identification, and index into other sections of the file.

[0471] Core Header Structure

[0472] AibVersion [4 bytes]: Magic number or appInstallBlock version number (which identifies the version of the appInstallBlock structure rather than the contents).

[0473] AppId [16 bytes]: this is an application identifier unique for each application. On Windows, this identifier is the GUID generated from the 'guidgen' program. AppId for Word on Win98 will be different from Word on WinNT if it turns out that Word binaries are different between NT and 98.

[0474] VersionNo [4 bytes]: Version number. This allows us to inform the client that the appInstallBlock has changed for a particular appId. This is useful for changes to the AppInstallBlock due to minor patch upgrades in the application.

[0475] ClientOSBitMap [4 bytes]: Client OS supported bitmap or ID: for Win2K, Win98, WinNT (and generally for other and multiple OSs).

[0476] ClientOSServicePack [4 bytes]: For optional storage of the service pack level of the OS for which this appInstallBlock has been created. Note that when this field is set, the multiple OS bits in the above field ClientOSBitMap are not used.

[0477] Flags [4 bytes]: Flags pertaining to AppInstallBlock

[0478] Bit **0**: Reboot—If set, the SAS client needs to reboot the machine after installing the AppInstallBlock on the client machine.

[0479] Bit **1**: Unicode—If set, the string characters are 2 bytes wide instead of 1 byte.

[0480] HeaderSize [2 bytes]: Total size in bytes of the header section.

[0481] Reserved [32 bytes]: Reserved spaces for future.

[0482] NumberOfSections [1 byte]: Number of sections in the index table.

[0483] This determines the number of entries in the index table structure described below:

[0484] Index Table Structure: (variable number of entries)

[0485] SectionType [1 bytes]: The type of data described in section. 0=file section, 1=variable section, 2=prefetch section, 3=profile section, 4=comment section, 5=code section.

[0486] SectionOffset [4 bytes]: The offset from the beginning of the file indicates the beginning of section.

[0487] SectionSize [4 bytes]: The size in bytes of section.

[0488] Variable Structure

[0489] ApplicationNameIsAnsi [1 byte]: 1 if ansi, 0 if Unicode.

[0490] ApplicationNameLength [4 bytes]: Byte size of the application name

[0491] ApplicationName [X bytes]: Null terminating name of the application

[0492] 2. File Section

[0493] The file section **3102** contains a subset of the list of files needed by the application to run properly. This section does not enumerate files located in the standard application program directory. It consists of information about files copied into an 'unusual' directory during the installation of an application. If the file content is small (typically less than 1 MB), the file is copied to the client machine. Otherwise, the file is relocated to the standard program directory suitable for streaming. The file section data is a list of trees stored in a contiguous sequence of address spaces according to the pre-order traversal of the trees. A node in the tree can correspond to one or more levels of directories. A parent-child node pair is combined into a single node if the parent node has only a single child. Parsing the tree from the root of the tree to a leaf node results in a fully legal Windows pathname including the drive letter. Each entry of the node in the tree consists of the following structure:

[0494] Directory Structure: (variable number of entries)

[0495] Flags [**4** byte]: Bit **0** is set if this entry is a directory

[0496] NumberOfChildren [2 bytes]: Number of nodes in this directory

[0497] DirectoryNameLength [4 bytes]: Length of the directory name

[0498] DirectoryName [X bytes]: Null terminating directory name

[0499] Leaf Structure: (variable number of entries)

[0500] Flags [4 byte]: Bit **1** is set to 1 if this entry is a spoof or copied file name

[0501] FileVersion [8 bytes]: Version of the file GetFileVersionInfo( ) if the file is win32 file image. Need variable file version size returned b y GetFileVersioninfoSize( ). Otherwise use file size or file modified time to compare which file is the later version.

[0502] FileNameLength [4 bytes]: Byte size of the file name

[0503] FileName [X bytes]: Null terminating file name

[0504] DataLength [4 bytes]: Byte size of the data. If spoof file, then data is the string of the spoof directory. If copied file, then data is the content of the file

[0505] Data [X bytes]: Either the spoof file name or the content of the copied file

[0506] 3. Add Variable and Remove Variable Sections

[0507] The add and remove variable sections **3103** contain the system variable changes needed to run the application. In a Windows system, each section consists of several number of registry subtrees. Each tree is stored in a contiguous sequence of address spaces according to the pre-order traversal of the tree. A node in the tree can correspond to one or more levels of directory in the registry. A parent-child node pair is combined into a single node if the parent node has only a single child. Parsing the tree from the root of the tree to a leaf node results in a fully legal key name. The order of the trees is shown here.

[0508] a. Registry Subsection:

[0509] 1. "HKCR": HKEY_CLASSES_ROOT

[0510] 2. "HKCU": HKEY_CURRENT_USER

[0511] 3. "HKLM": HKEY_LOCAL_MACHINE

[0512] 4. "HKUS": HKEY_USERS

[0513] 5. "HKCC": HKEY_CURRENT_CONFIG

[0514] Tree Structure: (5 entries)

[0515] ExistFlag [1 byte]: Set to 1 if this tree exist, 0 otherwise.

[0516] Key or Value Structure entries [X bytes]: Serialization of the tree into variable number key or value structures described below.

[0517] Key Structure: (variable number of entries)

[0518] KeyFlag [1 byte]: Set to 1 if this entry is a key or 0 if it's a value structure

[0519] NumberOfSubchild [4 bytes]: Number of subkeys and values in this key directory

[0520] KeyNameLength [4 bytes]: Byte size of the key name

[0521] KeyName [X bytes]: Null terminating key name

[0522] Value Structure: (variable number of entries)

[0523] KeyFlag [1 byte]: Set to 1 if this entry is a key or 0 if it's a value structure

[0524] ValueType [4 byte]: Type of values from the Win32 API function RegQueryValueEx( ): REG_SZ, REG_BINARY, REG DWORD, REG_LINK, REG_NONE, etc . . .

[0525] ValueNameLength [4 bytes]: Byte size of the value name

[0526] ValueName [X bytes]: Null terminating value name

[0527] ValueDataLength [4 bytes]: Byte size of the value data

[0528] ValueData [X bytes]: Value of the Data

[0529] In addition to registry changes, an installation in a Windows system may involve changes to the ini files. The following structure is used to communicate the ini file changes needed to be done on the SAS client machine. The ini entries are appended to the end of the variable section after the five registry trees are enumerated.

[0530] b. INI Subsection:

[0531] NumFiles [4 bytes]: Number of INI files modified.

[0532] File Structure: (variable number of entries)

[0533] FileNameLength [4 bytes]: Byte length of the file name

[0534] FileName [X bytes]: Name of the INI file

[0535] NumSection [4 bytes]: Number of sections with the changes

[0536] Section Structure: (variable number of entries)

[0537] SectionNameLength [4 bytes]: Byte length of the section name

[0538] SectionName [X bytes]: Section name of an INI file

[0539] NumValues [4 bytes]: Number of values in this section

[0540] Value Structure: (variable number of entries)

[0541] ValueLength [4 bytes]: Byte length of the value data

[0542] ValueData [X bytes]: Content of the value data

[0543] 4. Prefetch Section

[0544] The prefetch section **3104** contains a list of file blocks. The Builder profiler determines the set of file blocks critical for the initial run of the application. This data includes the code to start and terminate the application. It includes the file blocks containing frequently used commands. For example, opening and saving of documents are frequently used commands and should be prefetched if possible. Another type of block to include in the prefetch section are the blocks associated with frequently accessed directories and file metadata in this directory. The prefetch section is divided into two subsections. One part contains the critical blocks that are used during startup of the streamed application. The second part consists of the blocks accessed for common user operations like opening and saving of document. The format of the data is described below:

[0545] a. Critical Block Subsection:

[0546] NumCriticalBlocks [4 bytes]: Number of critical blocks.

[0547] Block Structure: (variable number of entries)

[0548] FileNumber [4 bytes]: File Number of the file containing the block to prefetch

[0549] BlockNumber [4 bytes]: Block Number of the file block to prefetch

[0550] b. Common Block Subsection:

[0551] NumCommonBlocks [4 bytes]: Number of critical blocks.

[0552] Block Structure: (variable number of entries)

[0553] FileNumber [4 bytes]: File Number of the file containing the block to prefetch

[0554] BlockNumber [4 bytes]: Block Number of the file block to prefetch

[0555] 5. Profile Section

[0556] The profile section **3105** consists of a reference sequence of file blocks accessed by the application at runtime. Conceptually, the profile data is a two dimensional matrix. Each entry [row, column] of the matrix is the frequency, a block row is followed by a block column. In any realistic applications of fair size, this matrix is very large and sparse. The proper data structure must be selected to store this sparse matrix efficiently in required storage space and minimize the overhead in accessing this data structure access.

[0557] The section is constructed from two basic structures: row and column structures. Each row structure is followed by N column structures specified in the NumberColumns field. Note that this is an optional section. But with appropriate profile data, the SAS client prefetcher performance can be increased.

[0558] Row Structure: (variable number of entries)

[0559] FileNumber [4 bytes]: File Number of the row block

[0560] BlockNumber [4 bytes]: Block Number of the row block

[0561] NumberColumns [4 bytes]: number of blocks that follows this block. This field determines the number of column structures following this field.

[0562] Column Structure: (variable number of entries)

[0563] FileNumber [4 bytes]: File Number of the column block

[0564] BlockNumber [4 bytes]: Block Number of the column block

[0565] Frequency [4 bytes]: frequency the row block is followed by column block

[0566] 6. Comment Section

[0567] The comment section **3106** is used by the Builder to describe this AppInstallBlock in more detail.

[0568] CommentLengthisAnsi [1 byte]: 1 if string is ansi, 0 if Unicode format.

[0569] CommentLength [4 bytes]: Byte size of the comment string

[0570] Comment [X bytes]: Null terminating comment string

[0571] 7. Code Section

[0572] The code section **3107** consists of the application-specific initialization code needed to run on the SAS client to setup the client machine for this particular application. This section may be empty if the default initialization procedure in the SAS client is able to setup the client machine without requiring any application-specific instructions. On the Windows system, the code is a DLL file containing two exported function calls: Install( ), Uninstall ( ). The SAS client loads the DLL and invokes the appropriate function calls.

[0573] CodeLength [4 bytes]: Byte size of the code

[0574] Code [X bytes]: Binary file containing the application-specific initialization code. On Windows, this is just a DLL file.

[0575] 8. LicenseAgreement Section

[0576] The Builder creates the license agreement section **3108**. The SAS client displays the license agreement text to the end-user before the application is started for the first time. The end-user must agree to all licensing agreement set by the software vendor in order to use the application.

[0577] LicenseTextisAnsi [1 byte]: 1 if ansi, 0 if Unicode format.

[0578] LicenseTextLength [4 bytes]: Byte size of the license text o LicenseAgreement [X bytes]: Null terminating license agreement string

CLIENT INSTALLATION AND EXECUTION OF STREAMED APPLICATIONS

[0579] Summary

[0580] This section describes the process of installing and uninstalling streamed application on the client machine. With respect to **FIG. 32**, the lifecycle of the Application Install Block is shown. The Application Stream Builder **3202** takes original application files **3201** and produces a corresponding Application Install Block and Stream Application Set **3203**. These two files get installed onto the application servers **3206**. On the right side of the drawing, it shows how either the administrator or the user can subscribe to the application from either the client computer **3208** or an administration computer **3207**. Once the user logons onto the client computer **3208**, the license and the AIB **3203** are acquired from the license **3205** and application servers **3206**, respectively.

[0581] The following are features of a preferred embodiment of the invention:

[0582] 1. The streamed application installation process installs just the description of the application, not the total content of the application. After installing such description, the client system looks and feels similar to having installed the same app using a non-streamed method. This has the following benefits:

[0583] a. Takes a very small fraction of the time to install the application.

[0584] b. Takes a very small fraction of the disk space.

[0585] c. Client does not have to wait for the entire application to be downloaded. This is particularly important to users with slow network connections.

[0586] The application description is subsequently un-installed without requiring deleting the total contents of the application. This has the benefit that it takes a very small fraction of the time to uninstall the application.

[0587] 2. Enhancing streamed application's performance by:

[0588] a. Copying small portions of application's code and data (pages) that are critical to performance.

[0589] b. Providing client with the initial profile data that can be used to perform pre-fetching.

[0590] This has the following benefits:

[0591] 1. User experiences smooth and predictable application launch.

[0592] 2. Scalability of Application servers increases by reducing the number of client connections.

[0593] 3. An administrator can arrange applications to be installed automatically on client computers. Administrator can also arrange the installation on various client computers simultaneously without being physically present on each client computer. This has the following benefits:

[0594] a. Users are not burdened with the process of installing streamed applications.

[0595] b. Reduced administration expense.

[0596] Overview of Components Relevant to the Install Process

[0597] Subscription Server **3204**: allows users to create accounts & to rent.

[0598] License Server **3205**: authenticates users & determines licensing rights to applications.

[0599] Application Server **3206**: provides application bits to licensed users securely & efficiently.

[0600] Application Install Manager—a component installed on the streaming client that is responsible for installing and uninstalling streamed applications.

[0601] Application Install Block (AIB) **3203**—a representation of what gets installed on the client machine when a streamed application is installed. It contains portions of the application that are responsible for registering the application with the client operating system and other data that enhances the execution of streamed application.

[0602] Application Stream Builder **3202**—preprocesses apps & prepares files to be installed on Application Server and data, such as AIB, to be installed by Client Application Installer.

[0603] Stream Application Set **3203**—a method of representing the total content of the application in a format that is optimal for streaming.

[0604] Client Streaming File System—integrates client exec with paging from a special file system backed by remote network-accessed server-based store

[0605] Application Install Block (AIB)

[0606] Installing and un-installing a stream application requires an understanding of what AIB is and how it gets manipulated by the various components in the overall streaming system. AIB is physically represented as a data file with various different sections. Its contents include:

[0607] Streamed application name and identification number.

[0608] Software License Agreement.

[0609] Registry spoof set.

[0610] File spoof set.

[0611] Small number of application pages—initial cache contents.

[0612] Application Profile Data.

[0613] AIB Lifecycle

[0614] The following describes the AIB lifecycle:

[0615] 1. Using the process described in the section above concerning converting apps for stream delivery and subsequent execution, an application install block is created by the Application Stream Builder. Initially, there will be one AIB per application, however, as the application evolves via patches and service packs, new AIBs may need to be generated.

[0616] 2. Using a process described in the section above regarding server-side performance optimization, AIB will get hosted by the application servers.

[0617] 3. "Subscribing" the application by communicating with the subscription server. Subscribing to an application requires a valid account with the ASP. Either the user or an administrator acting on the user's behalf can subscribe the application. In addition, the application can be subscribed to from any computer on the Internet, not just the client machine where the application will be eventually installed. This allows an administrator to subscribe applications for a group of users without worrying about individual client machines.

[0618] 4. The client machine acquires the license for the application from the license server. If the application was subscribed from the client machine itself, this step will happen automatically after subscribing to the application. If the subscription happened from a different machine, e.g., the administrator's machine, this step will happen when the user logs on the client machine. As an acknowledgment of having a valid license, the license server gives the client an encrypted access token.

[0619] 5. Fetch the contents of AIB from the application server. This step is transparent and happens immediately after the preceding step. Since application server requires the client to possess a valid access token, it ensures that only subscribed and licensed users can install the streamed application.

[0620] 6. The Application Install Manager (AIM) performs the act of installing the application information, as specified by the AIB, on the client system.

[0621] Installing a Streamed Application

[0622] AIM downloads AIB from the application server and takes the necessary steps in installing the application description on the client system. It extracts pieces of information from AIB and sends messages to various other components (described later) to perform the installation. AIM also creates an Install-Log that can be used when un-installing the streamed application.

[0623] 1. Display a license agreement to the user and wait for the user to agree to it.

[0624] 2. Extract File Spoof Data and communicate that to the Client File Spoofer. The list of files being spoofed will be recorded in the Install-Log.

[0625] 3. Extract Registry Spoof Data and communicate that to the Client Registry Spoofer. The list of Registries being spoofed will be recorded in the Install-Log.

[0626] 4. Extract Initial Cache Content and communicate that to the Client Prefetch Unit.

[0627] 5. Extract Profile Data and communicate that to the Client Prefetch Unit.

[0628] 6. Save the Install-Log in persistent storage.

[0629] Un-Installing a Streamed Application

[0630] Un-installation process relies on the Install-Log to know what specific items to un-install. Following steps are performed when un-installing and application:

[0631] 1. Communicate with the Client Registry Spoofer to remove all registries being spoofed for the application being un-installed.

[0632] 2. Communicate with the Client File Spoofer to disable all files being spoofed for the application being un-installed.

[0633] 3. Communicate with the Client Prefetch Unit to remove all Profile Data for the application being un-installed.

[0634] 4. Communicate with the Client Cache Manager to remove all pages being cached for the application being un-installed.

[0635] 5. Delete the Install-Log.

[0636] Client File Spoofer

[0637] A file spoofer component is installed on the client machine and is responsible for redirecting file accesses from a local file system to the streaming file system. The spoofer operates on a file spoof database that is stored persistently on the client system; it contains a number of file maps with following format:

[0638] [Original path of a local file]←→[New path of a file on streaming drive]

[0639] Where "←→" indicates a bidirectional mapping between the two sides of the relationship shown.

[0640] When a streamed application is installed, the list of new files to spoof (found in AIB) is added to the file spoof database. Similarly, when a streamed application is un-installed, a list of files being spoofed for that application is removed from the file spoof database.

[0641] On clients running the Windows 2000 Operating System, the file spoofer is a kernel-mode driver and the spoof database is stored in the registry.

[0642] Client Registry Spoofer

[0643] The Registry Spoofer intercepts all registry calls being made on the client system and re-directs calls manipulating certain registries to an alternate path. Effectively, it is mapping the original registry to an alternate registry transparently. Similar to the client file spoofer, the registry spoofer operates on a registry spoof database consisting entries old/new registry paths. The database must be stored in persistent storage.

[0644] When a streamed application is installed, the list of new registries to spoof (found in AIB) is added to the registry spoof database. Upon un-installation of a streamed application, its list of spoofed registries is removed from the registry spoof database.

[0645] On clients running the Windows 2000 Operating System, the registry spoofer is a kernel-mode driver and the registry spoof database is stored in the registry.

[0646] Client Prefetch Unit

[0647] In a streaming system, it is often a problem that the initial invocation of the application takes a lot of time because the necessary application pages are not resent on the client system when neeeded. A key aspect of the client install is that by using a client prefetch unit, a system in accordance with the present invention significantly reduces the performance hit associated with fetching. The Client Prefetch Unit performs two main tasks:

[0648] 1. Populate Initial Cache Content,

[0649] 2. Prefetch Application Pages.

[0650] Initial Cache Content

[0651] The Application Stream Builder determines the set of pages critical for the initial invocation and packages them as part of the AIB. These pages, also known as initial cache content, include:

[0652] Pages required to start and stop the application,

[0653] Contents of frequently accessed directories,

[0654] Application pages performing some of the most common operations within application. For example, if Microsoft Word is being streamed, these operations include: opening & saving document files & running a spell checker.

[0655] When the Stream Application is installed on the client, these pages are put into the client cache; later, when the streamed application is invoked, these pages will be present locally and network latency is avoided.

[0656] In preparing the Prefetch data, it is critical to manage the trade off of how many pages to put into AIB and what potential benefits it brings to the initial application launch. The more pages that are put into prefetch data, the smoother the initial application launch will be; however, since the AIB will get bigger (as a result of packing more pages in it), users will have to wait longer when installing the streamed application. In a preferred embodiment of the invention, the size of the AIB is limited to approximately 250 KB.

[0657] In an alternative embodiment of the invention the AIB initially includes only the page/file numbers and not the pages themselves. The client then goes through the page/file numbers and does paging requests to fetch the indicated pages from the server.

[0658] Prefetch Application Pages

[0659] When the streaming application executes, it will generate paging requests for pages that are not present in the client cache. The client cache manager must contact the application server and request the page in question. The invention takes advantage of this opportunity to also request additional pages that the application may need in the future. This not only reduces the number of connections to the application server, and overhead related to that, but also hides the latency of cache misses.

[0660] The application installation process plays a role in the pre-fetching b y communicating the profile data present in the AIB to the Client Prefetch Unit when the application is installed. Upon un-installation, profile data for the particular application will be removed.

CACHING OF STREAMED APPLICATION PAGES WITHIN THE NETWORK

[0661] Summary

[0662] This section describes how collaborative caching is employed to substantially improve the performance of a client server system in accordance with the other aspects of the present invention. Specifically, particular caching configurations and an intelligent way to combine these caching configurations are detailed.

[0663] Collaborative Caching Features

[0664] Using another client's cache to get required pages/packets (Peer Caching)

[0665] Using an intermediate proxy or node to get required pages/packets (Proxy Caching)

[0666] Using a broadcasting or multicasting mechanism to make a request (Multicast)

[0667] Using a packet based protocol to send requested pages/packets rather than a stream based one. (Packet Protocol)

[0668] Using concurrency to request a page through all three means (Peer Caching or Proxy Caching or the actual server) to improve performance (Concurrent Requesting).

[0669] Using heuristical algorithms to use all three ways to get the required pages (Smart Requesting).

[0670] These features have the following advantages:

[0671] These ideas potentially improve the performance of the client, i.e., they reduce the time a client takes to download a page (Client Performance).

[0672] These ideas improve the scalability of the server because the server gets fewer requests, i.e., requests which are fulfilled by a peer or a proxy don't get sent to the server. (Server Scalability)

[0673] These allow a local caching mechanism without needing any kind of modification of local proxy nodes or routers or even the servers. The peer-to-peer caching is achieved solely through the co-operation of two clients. (Client Only Implementation)

[0674] These ideas allow a client to potentially operate "offline" i.e., when it is not getting any responses from the server (Offline Client Operation).

[0675] These ideas allow the existing network bandwidth to be used more effectively and potentially reduce the dependency of applications on higher bandwidth (Optimal Use of Bandwidth).

[0676] These ideas when used in an appropriate configuration allow each client to require a smaller local cache but without substantially sacrificing the performance that you get by local caching. An example is when each client "specializes" in caching pages of a certain kind, e.g., a certain application. (Smaller Local Cache).

[0677] These ideas involve new interrelationships—peer-to-peer communication for cache accesses; or new configurations—collaborative caching. The reason this is called collaborative is because a group of clients can collaborate in caching pages that each of them needs.

[0678] Aspects of Collaborative Caching

[0679] 1. Peer Caching: A client X getting its pages from another client Y's local cache rather than its (X's) own or from the server seems to be a new idea. Major advantages: client performance, server scalability, client only implementation, offline client operation, optimal use of bandwidth, smaller local cache.

[0680] 2. Proxy Caching: The client getting its pages from an intermediate proxy which either serves the page from the local cache or from another intermediate proxy or the remote server (if none of the intermediate proxies has the page) is unique, at a minimum, for the pages of a streamed application. Major advantages: client performance, server scalability, offline client operation (to some extent), optimal use of bandwidth, smaller local cache.

[0681] 3. Multicast: Using multicasting (or selective broadcasting) considerably reduces peer-to-peer communication. For every cache request there is only one packet on the network and for every cache response there is potentially only one packet on the network in some configurations. This definitely helps reduce network congestion. Major advantages: client performance, server scalability, client only implementation, offline client operation, optimal use of bandwidth

[0682] 4. Packet Protocol: Because only datagram packets are used to request or respond to cache pages this saves the overhead of opening stream-based connections such as a TCP connection or an HTTP connection. Major advantages: client performance, client only implementation, offline client operation, and optimal use of bandwidth.

[0683] 5. Concurrent Requesting: If concurrent or intelligently staggered requests through all three means are issued to request a single page, the client will be able to receive the page through the fastest means possible for that particular situation. Major advantages: client performance, server scalability, offline client operation, and optimal use of bandwidth

[0684] 6. Smart Requesting: An adaptive or "smart" algorithm can be used to further enhance the overall performance of the client-server system. In this algorithm, the client uses the data of how past requests were processed to "une" new requests. For example, if the client's past requests were predominantly served by another client, i.e., Peer Caching worked, then for new page requests the client would first try to use Peer Caching, and wait some time before resorting to either Proxy Caching or direct server request. This wait time can again be calculated in an adaptive fashion. Major advantages: client performance, server scalability, client only implementation, offline client operation, and optimal use of bandwidth.

[0685] The concepts illustrated herein can be applied to many different problem areas. In all client-server implementations where a server is serving requests for static data, e.g., code pages of a streamed application or static HTML pages from a Website, the approaches taught herein can be applied to improve the overall client-server performance. Even if some of the protocols or configurations described in this document are not supported by the underlying network, it does not preclude the application of other ideas described herein that do not depend on such features. For example, if multicast (or selective broadcast) is not supported, ideas such as Concurrent Requesting or Smart Requesting can still be used with respect to multiple servers instead of the combination of a server, peer, and proxy. Also the use of words like Multicast does not restrict the application of these ideas to multicast based protocols. These ideas can be used in all those cases where a multicast like mechanism, i.e., selective broadcasting is available. Also note that the description of these ideas in the context of LAN or intranet environment does not restrict their application to such environments. The ideas described here are applicable to any environment where peers and proxies, because of their network proximity, offer significant performance advantages by using Peer Caching or Proxy Caching over a simple client-server network communication. In that respect, the term LAN or local area network should be understood to mean more generally as a collection of nodes that can communicate with each other faster than with a node outside of that collection. No geographical or physical locality is implied in the use of the term local area network or LAN.

[0686] Peer Caching

[0687] Referring to **FIG. 33**, how multiple peers collaborate in caching pages that are required by some or all of them is shown.

[0688] The main elements shown are:

[0689] Client **13301** through Client **63306** in an Ethernet LAN **3310**.

[0690] Router **1** and the local proxy serving as the Internet gateway **3307**. Note that it does not matter whether Router **1** and the proxy are one computer or two different ones.

[0691] Other routers from router **2** through router N **3308** that are needed to connect the LAN **3310** to the Internet **3311**.

[0692] A remote server **3309** (that is reachable only by going over the Internet **3311**) that is serving the pages that the above mentioned clients need.

[0693] A cloud that symbolizes the complexity of the Internet **3311** and potentially long paths taken by packets.

[0694] Client **23302** needs a page that it does not find in its local cache. It then decides to use the mechanism of Peer Caching before attempting to get the page from the local proxy (or the actual server through the proxy). The actual sequence of events is as follows:

[0695] 1. Client **23302** sends a request for the page it needs. This request is sent as a multicast packet to a predetermined multicast address and port combination. Lets call this multicast address and port combination as M.

[0696] 2. The multicast packet is received by all the clients that have joined the group M. In this case all six clients have joined the group M.

[0697] 3. Client **53305** receives the request and it records the sender's, i.e., Client **2**'s **3302**, address and port combination. Let's assume this address and port combination is A. Client **53305** processes the request and looks up the requested page in its own cache. It finds the page.

[0698] 4. Client **53305** sends the page to address A (which belongs to Client **23302**) as a packet.

[0699] 5. Client **23302** receives the page it needs and hence does not need to request the server for the page.

[0700] Proxy Caching

[0701] With respect to **FIG. 43, a** transparent proxy and how clients use it to get pages is shown. Again the elements here are the same as in the previous figure:

[0702] Client **13401** through Client **63406** in an Ethernet LAN **3410**.

[0703] Router **1** and the local proxy serving as the Internet gateway **3407**. Note that it does not matter whether Router 1 and the proxy are one computer or two different ones.

[0704] Other routers from router **2** through router N **3408** that are needed to connect the LAN **3410** to the Internet **3411**.

[0705] A remote server **3409** (that is reachable only by going over the Internet **3411**) that is serving the pages that the above mentioned clients need.

[0706] A cloud that symbolizes the complexity of the Internet **3411** and potentially long paths taken by packets.

[0707] Assume Peer Caching is either not enabled or did not work for this case. When Client **23402** needs a page, it makes a request to the proxy **3407**. The proxy **3407** finds the page in its local cache and returns it to Client **23402**. Because of this, the request did not go to the remote server **3409** over the Internet **3411**.

[0708] Multicast and Packet Protocol within a LAN

[0709] Referring to **FIG. 35**, the role played by multicast and unicast packets in Peer Caching is shown. The example of the drawing "Peer Caching" is used to explain multicast. Here Client **23502** has the IP address 10.0.0.2 and it opens port **3002** for sending and receiving packets. When Client **23502** needs a page and wants to use Peer Caching to get it, it forms a request and sends it to the multicast address and port 239.0.0.1:2001. All the other clients in the LAN **3508** that support Peer Caching have already joined the group 239.0.0.1:2001 so they all receive this packet.

[0710] Client **53505** receives this packet and it records the sender address (10.0.0.2:3002 in this case). It looks up the requested page and finds it in its local cache. It sends the page as a response packet to the address 1.0.0.0.2:3002.

[0711] Client **23502** receives this response packet since it was waiting at this port after sending the original multicast request. After ensuring the validity of the response, it retrieves the page it needs.

[0712] Note that more than one client can respond to the original multicast request.

[0713] However Client **23502** can discard all the later responses, since it has already received the page it needed.

[0714] Concurrent Requesting—Proxy First

[0715] With respect to **FIG. 36**, one particular case of how Concurrent Requesting is used is shown. This is a timeline of events that take place in the client. When a client first needs a page, it does not know whether it is going to get any responses through Peer Caching or not. Hence it issues a request to the proxy (or the server through the proxy) as soon as it needs the page. Then it issues a request using the Peer Caching mechanism. If there is indeed a peer that can return the page requested, the peer presumably could return the page faster than the proxy or the server. If this happens, the client may decide to use Peer Caching mechanism before attempting to get the page from the proxy or the server. The timeline essentially describes the following sequence of events:

[0716] 1. At time t=0, a page p is needed by the client **3601**.

[0717] 2. The client looks up its local cache, and it doesn't find page p.

[0718] 3. At time t=T1, it decides to send a request to the proxy to get the page **3602**.

[0719] 4. After a delay of amount Dp **3603**, at time t=T2 it also sends a request for the page p through the mechanism of Peer Caching **3604**. Note that D_P **3603** can be zero, in which case T1=T2.

[0720] 5. At time t=T3, a response is received from another peer that contains the page p that this client needs **3606**. Thus the response time of the Peer Caching mechanism is R_P=T3—T2 **3605**.

[0721] 6. At time t=T4, a response from the proxy/server is received that contains the page p **3608**. Hence the response time of the proxy/server is R_S=T4–T1 **3607**.

[0722] Note that since, $R_P<R_S$, the client will increase the weighting for Peer Caching in all of its future queries. That means it will decrease D_P, and if D_P is already zero, it will increase Ds (the delay before requesting proxy/server). On the other hand, if $R_P>R_S$ or if R_P were infinity, it will increase its weighting for proxy/server requesting. This is part of Smart Requesting that is explained elsewhere in this document.

[0723] Concurrent Requesting—Peer Caching First

[0724] Referring to **FIG. 37**, in contrast to the previous figure, the client has decided to use Peer Caching before requesting the proxy. So the sequence of events is as follows:

[0725] 1. At time t=0, a page p is needed by the client **3701**.

[0726] 2. The client looks up its local cache, and it doesn't find page p.

[0727] 3. At time t=T5, it decides to send a request for the page p through the mechanism of Peer Caching **3702**.

[0728] 4. After a delay of amount D_s **3703**, at time t=T6 it also sends a request for the page p to the proxy/server. Note that D_S can be zero, in which case T5=T6.

[0729] 5. At time t=T7, a response is received from another peer that contains the page p that this client needs **3706**. Thus the response time of the Peer Caching mechanism is R_P=T7–T5 **3705**.

[0730] 6. At time t=T8, a response from the proxy/server is received that contains the page p **3708**. Hence the response time of the proxy/server is R_S=T8–T6 **3707**.

[0731] As described in the previous drawing, the client increases the weighting of Peer Caching even more because it got a response through Peer Caching long before it got a response from the proxy/server. As a result of the increases weighting the delay D_s is increased even more.

[0732] Concurrent Requesting—Peer Caching Only

[0733] With respect to **FIG. 38**, in contrast with **FIG. 37**, the client has increased Ds **3805** (the delay before requesting a proxy/server) so much, that if a page is received before the expiry of the delay D_S **3805**, the client does not even make a request to the proxy/server. The shaded area **3806** shows the events that do not take place because of this.

[0734] Client-Server System with Peer and Proxy Caching

[0735] Referring to **FIG. 39, a** system level drawing that gives a system context for all the other figures and discussion in this document is shown. This drawing illustrates all three ways in which a client gets its page requests fulfilled. Note that:

[0736] Client **23902** gets its page request fulfilled through Peer Caching, i.e., multicast request.

[0737] Client **13901** gets its page request fulfilled through Proxy Caching, i.e., the proxy **3907** finds the page in its cache and returns it.

[0738] Client **33903** has to go to the server **3909** over the Internet **3908** to get its page request fulfilled.

[0739] Collaborative Caching Details

[0740] In a typical client-server model, caching could be used to improve the performance of clients and scalability of servers. This caching could be:

[0741] Local to the client where the client itself locally stores the pages it had received from the server in the past. Then the client would not need to request the proxy/server for any page that resides in the local cache as long as the locally cached page is "valid" from the server point of view.

[0742] On a proxy node that can be any node along the path taken by a packet that goes from the client to the server. The closer this proxy node is to the client the more improvement in the performance you get.

[0743] On a peer node, that is on another client. In this case, the two clients (the requesting client as well as the serving client) are on the same LAN or intranet, so that the travel time of a packet between the two nodes is considerably smaller as compared to the travel time of the packet from one of the clients to the server.

[0744] As far as caching is concerned, this section details the new ideas of Peer Caching and Proxy Caching. In addition, it also details the new ideas of Concurrent Requesting and Smart Requesting. The preferred approaches for implementing these ideas are also described here and these are Multicast and Packet Protocol.

[0745] The idea of Peer Caching is nothing but a client X taking advantage of the fact that a peer, e.g., say another client Y, on its LAN had, in the past, requested a page that X is going to request from its server. If the peer Y has that page cached locally on its machine, then X could theoretically get it much faster from Y than getting it from the server itself. If an efficient mechanism is provided for the two clients X and Y to collaborate on this kind of cache access, then that will offer many advantages such as: Client Performance, Server Scalability, Client Only Implementation, Offline Client Operation, Optimal Use of Bandwidth, Smaller Local Cache. Note that two clients were considered only as an example, the idea of Peer Caching is applicable to any number of peers on a LAN.

[0746] The idea of Multicast is to use the multicast protocol in the client making a Peer Caching request. Multicast can be briefly described as "selective broadcasting"—similar to radio. A radio transmitter transmits "information" on a chosen frequency, and any receiver (reachable by the transmitter, of course) can receive that information by tuning to that frequency. In the realm of multicast, the equivalent of a radio frequency is a multicast or class D IP address and port. Any node on the net can send datagram packets to a multicast IP address+port. Another node on the net can "join" that IP address+port (which is analogous to tuning to a radio frequency), and receive those packets. That node can also "leave" the IP address+port and thereby stop receiving multicast packets on that IP address+port.

[0747] Note that multicast is based on IP (Internet Protocol) and is vendor neutral. Also, it is typically available on the Ethernet LAN and, if routers supported it, it can also go beyond the LAN. If all the routers involved in a node's connection to the Internet backbone supported multicast routing, multicast packets theoretically could go to the whole Internet except the parts of the Internet that do not support multicast routing.

[0748] The use of multicast allows a client to not have to maintain a directory of peers that can serve its page requests. Also because of multicast there is only one packet per page request. Any peer that receives the request could potentially serve that request, so by using a multicast based request there are multiple potential servers created for a page request but only one physical packet on the network.

[0749] This contributes substantially in reducing network bandwidth, but at the same time increasing peer accessibility to all the peers. When implemented properly, the packet traffic due to Peer Caching will be proportional to the number of clients on the network participating in Peer Caching.

[0750] An idea related to Multicast is Packet Protocol. Note that Multicast itself is a packet-based protocol as opposed to connection based. The idea of Peer Caching here is described using Multicast and Packet Protocol. The Peer Caching request is sent as a multicast request and the response from a peer to such a request is also sent as a packet (not necessarily a multicast packet). Sending packets is much faster than sending data through a connection-based protocol such as TCP/IP, although using packet-based protocol is not as reliable as using connection-based one. The lack of reliability in Packet Protocol is acceptable since Peer Caching is used only to improve overall performance of the Client-Server system rather than as a primary mechanism for a client to get its pages. The underlying assumption made here is that a client could always get its pages from the server, if Peer Caching or Proxy Caching does not work for any reason.

[0751] The ideas of Concurrent Requesting and Smart Requesting describe how Peer Caching, Proxy Caching and client-server access could be combined in an intelligent fashion to achieve optimal performance of the whole Client-Server system. As part of Concurrent Requesting, a client is always prepared to make concurrent requests to get the page it needs in the fastest way possible. Concurrent Requesting would require the use of objects such as threads or processes that would allow one to programmatically implement Concurrent Programming. This document assumes the use of threads to describe a possible and preferred way to implement Concurrent Requesting.

[0752] The idea of Smart Requesting includes using an adaptive algorithm to intelligently stagger or schedule requests so that a client, even while using Concurrent Requesting, would not unnecessarily attempt to get a page through more than one means. An example of this is when a client has consistently gotten its page requests fulfilled through Peer Caching in the past. It would come to depend on Peer Caching for future page requests more than the other possible means. On the other hand, if Peer Caching has not worked for that client for some time, it would schedule a proxy request before a Peer Caching request. Smart Requesting involves dynamically calculating the delays D_P and D_S based how well Peer Caching and Proxy Caching has worked for the client. Please see **FIGS. 36 through 38**.

[0753] The following is an algorithmic description using pseudo-code of an illustrative embodiment.

[0754] startOurClient is a function that is invoked initially when the client is started.

```
void startOurClient() {
    Initialize the global variable delay to appropriate value based on a
    predfined policy. When delay is positive, it signifies the amount of time to
    wait after Proxy Caching before Peer Caching is attempted; and when
    delay is negative it signifies the amount of time to wait after Peer Caching
    before Proxy Caching is attempted. As an example:
    delay = 50;
    Start a thread for peer responses (i.e., Peer Caching server) with thread
    function as peerServer;
}
getPage function
The function getPage is called by the client's application to get a page. This
function looks up the local cache and if the page is not found, attempts to get the
page from a peer or proxy/server using the ideas of Concurrent Requesting and
Smart Requesting.
void getPage(PageIdType pageId) {
    if pageId present in the local cache then {
        retrieve it and return it to the caller;
    }
    if (delay > 0) {
        myDelay = delay;
        Call requestProxy(pageId);
    }
    else {
        myDelay = -delay;
        Call requestPeer(pageId);
    }
    Wait for gotPage event to be signaled for a maximum of myDelay
    milliseconds;
    If the page was obtained as indicated by gotPage being signaled {
        Modify delay appropriately i.e., if the page was obtained through
        Proxy Caching increment delay else decrement it;
        Return the page;
    }
    if (delay > 0) {
        Call requestPeer(pageId);
    }
    else {
        Call requestProxy(pageId);
    }
    Wait for the page to come through either methods;
    Depending on how the page came (through Proxy Caching or Peer
    Caching) increment or decrement delay;
    Return the page;
}
```

[0755] The function requestProxy sends a page request to the proxy and starts a thread that waits for the page response (or times out). The function proxyResponse is the thread function that waits for the response based on the arguments passed to it.

```
void requestProxy(pageId) {
    Send a page request for pageId to a predefined proxy/server as per the
    proxy/server protocol;
    Start a thread with the thread function proxyResponse that waits for
    the response to the request - the function proxyResponse is passed
    arguments: the socket X where it should wait and pageId.
}
```

-continued

```
void proxyResponse(socket X, pageId) {
     Wait at the socket X for a response with a timeout of time TY;
     If a response was received at socket X {
          Uncompress the packet if necessary;
          Validate the packet and ensure that this is a
          valid response to the request and has the page requested (he.,
          match the page Id);
     }
     else {
          // this is time out: didn't receive any
          // response in time TY
          Set appropriate indicator to indicate time-out;
     }
     Signal an event to signify completion of this thread;
}
requestPeer and peerResponse functions
```

[0756] The function requestPeer is similar to requestProxy except that it sends a page request to peers and starts a thread that waits for the page response (or times out). The function peerResponse is the thread function that waits for the response based on the arguments passed to it.

```
Void requestPeer(pageId) {
        Create a UDP socket X bound to port 3002;
     Compose a packet that consists of:
          a code indicating that this is a request for a page
          Some kind of an identifier that uniquely identifies the page
          wanted such as the URL
          other info such as security information or access validators
        Send this packet as a multicast packet to 239.0.0.1:2001 through
        the socket X created above;
     Create a thread with the thread function peerResponse and pass
     socket X and pageId as arguments to it;
}
Void peerResponse(socket X, pageId) {
     Wait at the socket X for a response with a timeout of time TX;
     If a packet was received at socket X {
          Uncompress the packet if necessary;
          Validate the packet and ensure that this is a
          valid response to the request and has the page requested (i.e.,
          match the pageId);
     }
     else {
     // this is time out: didn't receive any
     // response in time TX
     Set appropriate indicator to indicate time-out;
     }
     Signal an event to signify completion of this thread;
}
peerServer function
```

[0757] The function peerServer described below serves page requests received through Peer Caching as multicast packets. The function below describes how this thread would work:

```
void peerServer() {
     Create a multicast socket M bound to port 2001;
     Have M "join" the IP address 239.0.0.1;
     while (not asked to terminate) {
          Wait at M for a multicast packet;
          If a packet is received then {
     Store the source IP addr in S along with the source port number in B;
     Validate the packet that it is a valid request for a page that can be
     served (with valid security credentials);
```

-continued

```
               Look up the page id in the local client cache;
               If the page is found {
                    Compose a packet that contains the pageId of the
                    page as well as the page contents to send;
                    Optionally compress the packet before sending;
                    Send this packet to the IP address S at port B;
               }
          }
     }
}
```

PIRACY PREVENTION FOR STREAMED APPLICATIONS

[0758] Summary

[0759] The details presented in this section describe new techniques of the invention that have been developed to combat software piracy of applications provided over networks, in situations where an ASP's clients' machines execute the software applications locally. The remote ASP server must make all the files that constitute an application available to any subscribed user, because it cannot predict with complete accuracy which files are needed at what point in time. Nor is there a reliable and secure method by which the server can be aware of certain information local to the client computer that could be useful at stopping piracy. The process may be a rogue process intent on pirating the data, or it may be a secure process run from an executable provided by the ASP.

[0760] Aspects of the Invention

[0761] 1. Client-side fine-grained filtering of file accesses directed at remotely served files, for anti-piracy purposes. Traditional network filesystems permit or deny file access at the server side, not the client side. Here, the server provides blanket access to a given user to all the files that the user may need during the execution of an application, and makes more intelligent decisions about which accesses to permit or deny.

[0762] 2. Filtering of file accesses based on where the code for the process that originated the request is stored. Traditional file systems permit or deny file access usually based on the credentials of a user account or process token, not on where the code for the process resides. Here, a filesystem may want to take into account whether the code for the originating process resides in secure remote location or an insecure local location.

[0763] 3. Identification of crucial portions of served files and filtering file accesses depending on the portion targeted. The smallest level of granularity that traditional file systems can operate on is at the level of files, not at the level of the sections contained in the files (for example, whether or not data from a code section or a resource section is requested).

[0764] 4. Filtering of file accesses based on the surmised purpose of the file access, as determined by examining the program stack or flags associated with the request. Traditional file systems do not attempt to determine why a file access was issued before permitting or denying the access, e.g., whether the purpose is to copy the data or page in the data as code for execution.

[0765] 5. Filtering of file accesses based on the surmised purpose of the file access, as determined by examining a history of previous file accesses by the same process.

Traditional file systems do not keep around histories of which blocks a given requester had previously requested from a file. This history can be useful in seeing if the requests match a pattern that suggests a file copy is occurring as opposed to code execution.

[0766] Benefits of the Anti-Piracy Features of the Present Invention

[0767] This is an enabler technology that allows a programmer to build security into a certain type of application delivery system that would otherwise not be possible. Several companies are developing technology that allows an application to be served remotely, but executed locally. Current filesystems provide no way to protect the files that make up this application from being copied and thus pirated. The above techniques are tools that enable a filesystem to allow just those requests that will let the application run normally and block those that are the result of attempts to pirate the application's code or data. This provides a competitive advantage to those software providers who use this technology, because piracy results in lost revenue and, by preventing this, piracy they can prevent this loss.

[0768] The techniques described herein were developed for the purpose of preventing the piracy of computer software programs that are served from a remote server, but executed on a local client. However, they can be used by any computer software security solution that would benefit from the ability to filter file accesses with more flexibility than currently provided by most filesystems.

[0769] When a filesystem receives a request, it must decide whether or not the request should be granted or denied for security reasons. If the target file is local, the filesystem makes the decision by itself, and if the target file is remote, it must ask the server to handle the request for it. The above techniques are ways in which the filesystem can gather more information about the request than it would ordinarily have. It can then use that information to improve the quality of its decisions. Traditional approaches, such as granting a currently logged-in user access to certain files and directories that are marked with his credentials, are not flexible enough for many situations. As for remote files, the server has only a limited amount of information about the client machine. The filesystem at the client side can make grant/deny decisions based on local information before ever asking the server, in order to provide a more intelligent layer of security.

[0770] For example, it may be desirable to allow the user to execute these files, but not copy them. It may be desirable to grant access to only certain processes run b y the user, but not others, because it is judged that some processes to be more secure or well-behaved than others. And it may be desirable to allow the user to access only certain sections of these files and from only certain processes for certain periods of time. The above techniques are tools that are added to a filesystem to give it these abilities.

[0771] Overview of the Anti-Piracy Features of the Present Invention

[0772] With respect to **FIG. 40**, preventing piracy of remotely served, locally executed applications is shown. This figure illustrates the problem of software piracy in an application delivery system, and how it can be stopped using the techniques described in this section. The client computer **4001** is connected to a server **4009** run by an ASP **4007**. The server **4009** provides access to application files **4008**, out of which the application executable is run by the client **4001** locally on his machine. (This is Process #**14002**). However, the user can attempt to access and copy the application files to local storage **4009** on his machine, and thus b e able to run them without authorization or give them to another person. But since all requests directed at the remote files **4006** must first pass through the local network filesystem, this filesystem can be enhanced **4005** to deny all such requests that it thinks are the result of an attempt at piracy.

[0773] Referring to **FIG. 41**, the filtering of accesses to remote application files, illustrating New Technique #1, as described above is shown. (Note: the client computer represented here and in all subsequent figures is part of the same client-server system as in **FIG. 40**, but the server/ASP diagram has been omitted to save space.) A user **4102** who has been granted access to remotely served files **4106** representing an application is attempting to access these files. The local enhanced network filesystem **4103** is able to deny access to certain files **4105** and grant access to others **4104**, for the purpose of protecting critical parts of the application from piracy.

[0774] With respect to **FIG. 42**, the filtering of accesses to remote files based on process code location, illustrating New Technique #2, as described above, is shown. Here there are two processes on the client computer. Process #**14202** has been run from an executable file **4206** that is part of a remotely served application **4207**, and process #**24203** has been run from a local executable file **4204**. They are both attempting to access a remote data file **4206** that is part of the served application **4207**. The local enhanced network filesystem **4205** is denying Process #**24203** access and granting Process #**14202** access because Process #2's **4203** executable is stored locally, and thus is not secure, while Process #1's **4202** executable is provided by the server **4207**, and thus can be vouched for.

[0775] Referring to **FIG. 43**, the filtering of accesses to remote files based on targeted file section, illustrating New Technique #3, as described above, is shown. Here there is a single local process **4302** that is attempting to read from a remotely served executable file **4307**. The enhanced network filesystem **4304** is denying an attempt to read from the code section **4306** of the file **4307** while granting an attempt to read from a non-code section **4305** of the file **4307**. This is useful when access to some part of the file must be allowed, but access to other parts should be denied to prevent piracy of the entire file.

[0776] With respect to **FIG. 44**, the filtering of accesses to remote files based on surmised purpose, illustrating New Technique #4 as described above, is shown. Here, two attempts to read from the code section **4407** of a remote executable file **4408** are being made from a process **4402** that was run from this file **4408**. However, one request is denied because it originated **4406** from the process's code **4403** itself, while another is approved because it originated from code in the Virtual Memory Subsystem **4404**. This prevents even a rogue remote process from attempting to pirate its own code, while allowing legitimate requests for the code to be completed.

[0777] Referring to **FIG. 45**, the filtering of accesses to remote files based on past access history. illustrating New Technique #5 as described above, is shown. Here, two processes **4502**, **4503** run from a local executable **4504** are attempting to access a remote file **4508**. The enhanced network filesystem **4507** keeps around a history of previous file accesses by these processes **4505**, **4506**, which it consults to make decisions about permitting/denying further accesses. Process #1's **4502** access attempt is granted, while Process #2's **4503** is denied, because the filesystem **4507** detected a suspicious pattern in Process #2's **4503** previous access history **4506**.

[0778] Anti-Piracy Details of the Invention

[0779] Five anti-piracy embodiments are disclosed below that can be used by an ASP-installed network filesystem to combat piracy of remotely served applications. The ASP installs a software component on the client that is able to take advantage of local knowledge, e.g., which process on the client originated a request for data, and permit or deny requests for remote files before sending the requests to the server. That is, a network filesystem is installed on the local user's computer that manages access to these remote files. All input/output requests to these files must pass through this filesystem, and if the filesystem determines that a given request is suspicious in some way, it has the freedom to deny it.

[0780] Anti-Piracy Embodiment #1: Client-side Fine-grained Filtering of File Accesses Directed at Remotely Served Files, for Anti-piracy Purposes

[0781] Referring again to **FIG. 41**, the approach of the first anti-piracy embodiment is that a software component **4102** executing locally on a client computer **4101** has available to it much more information about the state of this computer than does a server providing access to remote files. Thus, the server can filter access only on a much coarser level that can this client component. An ASP can take advantage of this by installing a network filesystem **4103** on the client computer that is designated to handle and forward all requests directed at files located on a given remote server. This filesystem **4103** examines each request, and either grants or denies it depending on whether the request is justifiable from a security perspective. It can use information such as the nature of the originating process, the history of previous access by the process, the section of the targeted file being requested, and so on, in order to make its decision.

[0782] The best way known of implementing this approach is to write a network redirector filesystem component **4103** for the operating system that the ASP's clients' machines will be running. This component will be installed, and will make visible to the system a path that represents the server on which the ASP's application files are stored. The local computer can now begin accessing these files, and the filesystem **4103** will be asked to handle requests for these files. On most operating systems, the filesystem **4103** will register dispatch routines to the system that handle common file operations such as open, read, write and close. When a local process **4102** makes a request of an ASP-served file, the OS calls one of these dispatch routines with the request. In the dispatch routine, the filesystem **4103** examines the request and decides whether to deny it or grant it. If granted, it will forward the request to the remote server and send back the response to the operating system.

[0783] Anti-Piracy Embodiment #2: Filtering of File Accesses Based on Where the Code for the Process that Originated the Request is Stored

[0784] Referring again to **FIG. 42**, when a filesystem **4205** receives a request for access to a given file, the request always originates from a given process on the computer. By determining where the executable file that the process was run from is located, the network filesystem **4205** can make a more informed decision about the security risk associated with granting the request. For example, if the executable file **4204** is located on the local computer **4202**, then it may contain any code whatsoever, code that may attempt to copy and store the contents of any remote files it can gain access to. The filesystem **4205** can reject requests from these processes as being too risky. However, if the executable file **4206** is being served by the ASP's remote server **4207**, then the process can assume to be well-behaved, since it is under the control of the ASP. The filesystem **4205** can grant accesses that come from these processes **4202** in confidence that the security risks are minimal.

[0785] The best way known of implementing this approach is to modify a network filesystem **4205** to determine the identity of the process that originated a relevant open, read, or write request for a remote file. On some OSes a unique process ID is embedded in the request, and on others, a system call can be made to get this ID. Then, this ID must be used to look up the pathname of the executable file from which the process was run. To do this, upon initialization the filesystem **4205** must have registered a callback that is invoked whenever a new process is created. When this callback is invoked, the pathname to the process executable and the new process ID are provided as arguments, data which the filesystem **4205** then stores in a data structure. This data structure is consulted while servicing a file request, in order to match the process ID that originated the request with the process's executable. Then the root of the pathname of that executable is extracted. The root uniquely identifies the storage device or remote server that provides the file. If the root specifies an ASP server that is known to be secure, as opposed to a local storage device that is insecure, then the request can be safely granted.

[0786] Anti-Piracy Embodiment #3: Identification of Crucial Portions of Served Files and Filtering File Access Depending on the Portion Targeted

[0787] Referring again to **FIG. 43, a** served application usually consists of many files. In order to steal the application, a pirate would have to copy at least those files that store the code for the application's primary executable, and perhaps other files as well. This leads to the conclusion that some files are more important that others, and that some portions of some files are most important of all. Ordinarily, the best solution would be to deny access to the primary executable file and its associated executables in its entirety, but this is not usually possible. In order to initially run the application, the filesystem **4304** must grant unrestricted access to some portions of the primary executable. In order to prevent piracy, the filesystem **4304** can grant access selectively to just those portions that are needed. Additionally, the running application **4302** itself does not usually need to read its own code section, but does need to read other sections for purposes such as resource loading. Therefore, additional security can be introduced b y denying access to the code sections **4306** of ASP-served executables **4307** even to those executables themselves.

[0788] To implement this, modify a network filesystem's **4304** open file dispatch routine to detect when a remotely served executable **4307** is being opened. When this is detected, the executable file **4307** is examined to determine the offset and length of its code section **4306**, and this information is stored in a data structure. On most OSes, executable files contain headers from which this information can be easily read. In the read and write dispatch routines, the network filesystem **4304** checks if the request is for a remote executable **4307**, and if so, the offset and length of the code section **4306** of this executable **4307** is read from the data structure in which it was previously stored. Then the offset and length of the request are checked to see if they intersect the code section **4306** of this executable **4307**. If so, the request can be denied.

[0789] Anti-Piracy Embodiment #4: Filtering of File Accesses Based on the Surmised Purpose of the File Access, as Determined by Examining the Program Stack or Flags Associated with the Request

[0790] Referring again to **FIG. 44**, the approach of the fourth embodiment is that identical requests from the same process for a remotely served file can be distinguished based on the reason the request was issued. For example, on a computer with a virtual memory subsystem **4404**, the VMS's own code will be invoked to page-in code for a process that attempts to execute code in pages that are not currently present. To do this, the VMS **4404** must issue a read request to the filesystem **4405** that handles the process' **4402** executable file **4408**. Since this request is not for any ulterior purpose, such as piracy, and is necessary for the application to execute, the request should be granted. If the filesystem **4405** gets the originating process ID for such requests, the process whose code is being paged in will be known. However, this same process ID will also be returned for requests that originate as a result of an attempt by the process itself to read its own code (perhaps for the purpose of piracy). Many applications have loopholes that allow the user to execute a macro, for example, that reads and writes arbitrary files. If the filesystem **4405** simply filters requests based on process IDs, it will mistakenly allow users to pirate remotely served applications, as long as they can send the necessary reads and writes from within the remote application itself.

[0791] However, even if the process IDs are the same for two apparently identical requests, there are ways the filesystem **4405** can distinguish them. There are two ways to do this in a manner relevant to combating anti-piracy. The way to implement the first method is to have the filesystem **4405**, upon receiving a read request, check for the presence of the paging I/O flag that is supported b y several operating systems. If this flag is not present, then the request did not come from the VMS **4404**, but from the process itself **4403**, and thus the request is risky and not apparently necessary for the application to run. If the flag is present though, the request almost certainly originated from the VMS **4404** for the purpose of reading in code to allow the process to execute. The request should be allowed.

[0792] Another way to make this same determination is to have the filesystem **4405** examine the program stack upon receiving a read request. In several operating systems, a process will attempt to execute code that resides in a virtual page regardless of whether the page is present or not. If the page is not present, a page fault occurs, and a structure is placed onto the stack that holds information about the processor's current state. Then the VMS **4404** gets control. The V M S **4404** then calls the read routine of the filesystem **4405** that handles the process's executable file to read this code into memory. The filesystem **4405** now reads backwards up the stack up to a certain point, searching for the presence of the structure that is placed on the stack as a result of a page fault. If such a structure is found, the execution pointer register stored in the structure is examined. If the pointer is a memory address within the boundary of the virtual memory page that is being paged in, then the filesystem **4405** knows the read request is legitimate.

[0793] Anti-Piracy Embodiment #5: Filtering of File Accesses Based on the Surmised Purpose of the File Access, as Determined by Examining a History of Previous File Accesses by the Same Process

[0794] Referring again to **FIG. 45**, if one looks at the series of file requests that are typically made as a result of attempting to copy an executable file, as opposed to those made in the course of executing that file, one can see certain patterns.

[0795] The copy pattern is usually a sequence of sequentially ordered read requests, while the execution pattern tends to jump around a lot (as the result of code branches into non-present pages). A filesystem can be enhanced to keep around a history of requests made by specific processes on remotely served files. Then, for every subsequent request to such a file, the history for the originating process can be examined to check for certain patterns. If a file-copy pattern is seen, then the pirate may be attempting to steal the file, and the request should be denied. If an execution type pattern is seen, then the user is simply trying to run the application, and the request should be granted.

[0796] To implement this, a filesystem **4507** tells the operating system, via an operating system call, upon initialization, to call it back whenever a new process is created. When it is called back, the filesystem **4507** creates a new data structure for the process that will store file access histories **4505**, **4506**. Then, in its read-file dispatch routines, the filesystem **4507** determines the process ID of the originating process, and examines the process's access history **4505**, **4506**. It only examines entries in that history **4505**, **4506** that refer to the file currently being requested. It will then run a heuristic algorithm that tries to determine if the pattern of accesses more closely resembles an attempted file copy than code execution. An effective algorithm is to simply see if the past n read requests to this file have been sequential, where n is some constant. If so, then the request is denied. If not, then the request is granted. In either case, an entry is made to the filesystem's process access history **4505**, **4506** that records the file name, offset, and length of the request made by that process to this file.

CONCLUSION

[0797] Although the present invention has been described using particular illustrative embodiments, it will be understood that many variations in construction, arrangement and use are possible within the scope of this invention. Other embodiments may use different network protocols, different programming techniques, or different heuristics, in each component block of the invention. Specific examples of variations include:

- **[0798]** The proxy used in Proxy Caching could be anywhere in the Internet along the network path between a Client and the Server; and
- **[0799]** Concurrent Requesting and Smart Requesting can be implemented in hardware instead of software.

[0800] A number of insubstantial variations are possible in the implementation of anti-piracy features of the invention. For example, instead of modifying the filesystem proper to provide anti-piracy features, a network proxy component can be placed on the client computer to filter network requests made by a conventional local network filesystem. These requests generally correspond to requests for remote files made to the filesystem by a local process, and the type of filtering taught by the present invention can be performed on these requests. A filesystem filter component can also be written to implement these methods, instead of modifying the filesystem itself.

[0801] Although the invention is described herein with reference to the preferred embodiment, one skilled in the art will readily appreciate that other applications may be substituted for those set forth herein without departing from the spirit and scope of the present invention. Accordingly, the invention should only be limited by the Claims included below.

1. A process for client-side retrieval, storage, and execution of application programs and other related data streamed from a server across a computer network to a client system in a computer environment, comprising the steps of:

providing a streaming file system on said client;

wherein said streaming file system appears to said client to contain the installed application program;

wherein said streaming file system receives all requests from local processes for application program code or data that are part of the application;

providing a persistent cache on said client;

wherein said streaming file system satisfies requests for application program code or data by retrieving it from said persistent cache stored in a native file system or by retrieving it directly from said server; and

wherein application program code or data retrieved from said server is placed in said persistent cache for reuse.

2. The process of claim 1, wherein said persistent cache is encrypted with a key not permanently stored on said client to prevent unauthorized use or duplication of application code or data; and wherein said key is sent to said client upon application startup from said server and said key is not stored in the application program's persistent storage area in said persistent cache.

3. The process of claim 1, wherein said client initiates the prefetching of application program code and data from said server; and wherein said client inspects program code or data file requests and consults the contents of said persistent cache as well as historic information about application program fetching patterns and uses this information to request additional blocks of application program code and data from said server that said client expects will be needed soon.

4. The process of claim 1, wherein said server initiates the prefetching of application program code and data for said client; and wherein said server examines the patterns of requests made by said client and selectively returns to said client additional blocks that said client did not request but is likely to need soon.

5. The process of claim 1, further comprising the step of:

providing a client-to-client communication mechanism that allows local application customization to migrate from one client machine to another without involving server communication;

wherein when a user wishes to run an application on a second machine, but wishes to retain customizations made previously on the first, said client-to-client mechanism contacts the first machine to retrieve customized files and other customization data; and

wherein unmodified files are retrieved from said server.

6. The process of claim 1, further comprising the step of:

wherein said streaming file system is a copy-on-write file system that allows applications to write configuration or initialization files where they want to without rewriting the application, and without disturbing the local customization of other clients;

wherein each block of data in said persistent cache is marked as clean or dirty;

wherein pages marked as dirty have been customized by the application program and cannot be removed from the cache without losing client customization;

wherein pages marked as clean may be purged from the cache because they can be retrieved again from said server;

providing a cache index;

wherein said cache index indicates which pages in said persistent cache are clean and dirty; and

wherein said copy-on-write file system references said cache index to determine if a page is clean or dirty.

7. The process of claim 1, further comprising the step of: marking specific files in said persistent cache as not modifiable;

wherein said streaming file system does not allow any data to be written to said specific files that are marked as not modifiable; and

wherein attempts by any processes to mark any of said specific files as modifiable will not succeed.

8. The process of claim 1, further comprising the step of:

maintaining checksums of application code and data in said persistent cache;

wherein when a block of code or data is requested by a local process said streaming file system computes the checksum of the data block before it is returned to the local process; and

wherein if a computed checksum does not match the checksum stored in said persistent cache the cache entry is invalidated and a fresh copy of the page is retrieved from said server.

9. The process of claim 1, further comprising the step of:

assigning each file in an application program a unique identifier;

wherein files that are changed or added in an application upgrade are given new identifiers never before used for that application program;

wherein files that are unchanged retain the same number; and

wherein directories whose contents change are also considered changes. If any file changes, this will cause its parent to change, all the way up to the root directory.

10. The process of claim 9, wherein when an application upgrade occurs said client is given a new root directory for the application program by said server; wherein said new root directory is used by said streaming file system to search for files in the application program; wherein files that do not change can be reused from said persistent cache without downloading them again from said server; and wherein files with new identifiers are retrieved from said server.

11. The process of claim 9, wherein said application upgrades can be marked as mandatory by said server causing the new root directory for the application program to be used immediately; and wherein said streaming file system contacts said server when an application program is started in order to receive any application upgrades.

12. The process of claim 1, wherein said server broadcasts an application program's code and data and any client that is interested in that particular application program stores the broadcasted code and data for later use.

13. A process for client-side retrieval, storage, and execution of application programs and other related data streamed from a server across a computer network to a client system in a computer environment, comprising the steps of:

providing a kemel-mode streaming file system driver on said client;

providing a user-mode client on said client;

wherein said streaming file system receives all requests from local processes for application program code or data that are part of the application;

providing a persistent cache on said client;

wherein requests made to said streaming file system are directed to said user-mode client or retrieved from said persistent cache;

wherein said user-mode client handles the application program code and data streams from said server and sends the results back to said streaming file system driver; and

wherein application program code or data retrieved from said server is placed in said persistent cache for reuse.

14. The process of claim 13, wherein said persistent cache is encrypted with a key not permanently stored on said client to prevent unauthorized use or duplication of application code or data; and wherein said key is sent to said client upon application startup from said server and said key is not stored in the application program's persistent storage area in said persistent cache.

15. The process of claim 13, wherein said client initiates the prefetching of application program code and data from said server; and wherein said client inspects program code or data file requests and consults the contents of said persistent cache as well as historic information about application program fetching patterns and uses this information to request additional blocks of application program code and data from said server that said client expects will be needed soon.

16. The process of claim 13, wherein said server initiates the prefetching of application program code and data for said client; and wherein said server examines the patterns of requests made by said client and selectively returns to said client additional blocks that said client did not request but is likely to need soon.

17. The process of claim 13, further comprising the step of:

providing a client-to-client communication mechanism that allows local application customization to migrate from one client machine to another without involving server communication;

wherein when a user wishes to run an application on a second machine, but wishes to retain customizations made previously on the first, said client-to-client mechanism contacts the first machine to retrieve customized files and other customization data; and

wherein unmodified files are retrieved from said server.

18. The process of claim 13, further comprising the step of:

wherein said streaming file system is a copy-on-write file system that allows applications to write configuration or initialization files where they want to without rewriting the application, and without disturbing the local customization of other clients;

wherein each block of data in said persistent cache is marked as clean or dirty;

wherein pages marked as dirty have been customized by the application program and cannot be removed from the cache without losing client customization;

wherein pages marked as clean may be purged from the cache because they can be retrieved again from said server;

providing a cache index;

wherein said cache index indicates which pages in said persistent cache are clean and dirty; and

wherein said copy-on-write file system references said cache index to determine if a page is clean or dirty.

19. The process of claim 13, further comprising the step of:

marking specific files in said persistent cache as not modifiable;

wherein said streaming file system does not allow any data to be written to said specific files that are marked as not modifiable; and

wherein attempts by any processes to mark any of said specific files as modifiable will not succeed.

20. The process of claim 13, further comprising the step of:

maintaining checksums of application code and data in said persistent cache;

wherein when a block of code or data is requested by a local process said streaming file system computes the checksum of the data block before it is returned to the local process; and wherein if a computed checksum does not match the checksum stored in said persistent cache the cache entry is invalidated and a fresh copy of the page is retrieved from said server.

21. The process of claim 13, further comprising the step of:

assigning each file in an application program a unique identifier;

wherein files that are changed or added in an application upgrade are given new identifiers never before used for that application program;

wherein files that are unchanged retain the same number; and

wherein directories whose contents change are also considered changes. If any file changes, this will cause its parent to change, all the way up to the root directory.

22. The process of claim 21, wherein when an application upgrade occurs said client is given a new root directory for the application program by said server; wherein said new root directory is used by said streaming file system to search for files in the application program; wherein files that do not change can be reused from said persistent cache without downloading them again from said server; and wherein files with new identifiers are retrieved from said server.

23. The process of claim 21, wherein said application upgrades can be marked as mandatory by said server causing the new root directory for the application program to be used immediately; and wherein said streaming file system contacts said server when an application program is started in order to receive any application upgrades.

24. The process of claim 13, wherein said server broadcasts an application program's code and data and any client that is interested in that particular application program stores the broadcasted code and data for later use.

25. A process for client-side retrieval, storage, and execution of application programs and other related data streamed from a server across a computer network to a client system in a computer environment, comprising the steps of:

providing a streaming block driver on said client;

wherein said block driver provides the abstraction of a physical disk to a native file system already installed on the client operating system;

providing a persistent cache on said client;

wherein said block driver receives requests for physical block reads and writes from local processes which it satisfies out of said persistent cache on a standard file system that is backed by a physical disk drive; and

wherein requests that cannot be satisfied by said persistent cache are sent to said server.

26. The process of claim 25, wherein said persistent cache is encrypted with a key not permanently stored on said client to prevent unauthorized use or duplication of application code or data; and wherein said key is sent to said client upon application startup from said server and said key is not stored in the application program's persistent storage area in said persistent cache.

27. The process of claim 25, wherein said client initiates the prefetching of application program code and data from said server; and wherein said client inspects program code or data file requests and consults the contents of said persistent cache as well as historic information about application program fetching patterns and uses this information to request additional blocks of application program code and data from said server that said client expects will be needed soon.

28. The process of claim 25, wherein said server initiates the prefetching of application program code and data for said client; and wherein said server examines the patterns of requests made by said client and selectively returns to said client additional blocks that said client did not request but is likely to need soon.

29. The process of claim 25, further comprising the step of:

providing a client-to-client communication mechanism that allows local application customization to migrate from one client machine to another without involving server communication;

wherein when a user wishes to run an application on a second machine, but wishes to retain customizations made previously on the first, said client-to-client mechanism contacts the first machine to retrieve customized files and other customization data; and

wherein unmodified files are retrieved from said server.

30. The process of claim 25, further comprising the step of:

wherein said block driver is a copy-on-write block driver that allows applications to write configuration or initialization files where they want to without rewriting the application, and without disturbing the local customization of other clients;

wherein each block of data in said persistent cache is marked as clean or dirty;

wherein pages marked as dirty have been customized by the application program and cannot be removed from the cache without losing client customization;

wherein pages marked as clean may be purged from the cache because they can be retrieved again from said server;

providing a cache index;

wherein said cache index indicates which pages in said persistent cache are clean and dirty; and

wherein said copy-on-write block driver references said cache index to determine if a page is clean or dirty.

31. The process of claim 25, further comprising the step of:

marking specific files in said persistent cache as not modifiable;

wherein said block driver does not allow any data to be written to said specific files that are marked as not modifiable; and

wherein attempts by any processes to mark any of said specific files as modifiable will not succeed.

32. The process of claim 25, further comprising the step of:

maintaining checksums of application code and data in said persistent cache;

wherein when a block of code or data is requested by a local process said streaming file system computes the checksum of the data block before it is returned to the local process; and

wherein if a computed checksum does not match the checksum stored in said persistent cache the cache entry is invalidated and a fresh copy of the page is retrieved from said server.

33. The process of claim 25, further comprising the step of:

assigning each file in an application program a unique identifier;

wherein files that are changed or added in an application upgrade are given new identifiers never before used for that application program; wherein files that are unchanged retain the same number; and

wherein directories whose contents change are also considered changes.

If any file changes, this will cause its parent to change, all the way up to the root directory.

34. The process of claim 33, wherein when an application upgrade occurs said client is given a new root directory for the application program by said server; wherein said new root directory is used by said block driver to search for files in the application program; wherein files that do not change can be reused from said persistent cache without downloading them again from said server; and wherein files with new identifiers are retrieved from said server.

35. The process of claim 33, wherein said application upgrades can be marked as mandatory by said server causing the new root directory for the application program to be used immediately; and wherein said block driver contacts said server when an application program is started in order to receive any application upgrades.

36. The process of claim 25, wherein said server broadcasts an application program's code and data and any client that is interested in that particular application program stores the broadcasted code and data for later use.

37. A process for client-side retrieval, storage, and execution of application programs and other related data streamed from a server across a computer network to a client system in a computer environment, comprising the steps of:

providing a disk driver on said client;

providing a user mode client on said client;

wherein said disk driver sends all file requests that it receives to said user-mode client;

providing a persistent cache on said client; and

wherein said user-mode client attempts to satisfy said file requests from said program cache or by making requests from said server.

38. The process of claim 37, wherein said persistent cache is encrypted with a key not permanently stored on said client to prevent unauthorized use or duplication of application code or data; and wherein said key is sent to said client upon application startup from said server and said key is not stored in the application program's persistent storage area in said persistent cache.

39. The process of claim 37, wherein said client initiates the prefetching of application program code and data from said server; and wherein said client inspects program code or data file requests and consults the contents of said persistent cache as well as historic information about application program fetching patterns and uses this information to request additional blocks of application program code and data from said server that said client expects will be needed soon.

40. The process of claim 37, wherein said server initiates the prefetching of application program code and data for said client; and wherein said server examines the patterns of requests made by said client and selectively returns to said client additional blocks that said client did not request but is likely to need soon.

41. The process of claim 37, further comprising the step of:

providing a client-to-client communication mechanism that allows local application customization to migrate from one client machine to another without involving server communication;

wherein when a user wishes to run an application on a second machine, but wishes to retain customizations made previously on the first, said client-to-client mechanism contacts the first machine to retrieve customized files and other customization data; and

wherein unmodified files are retrieved from said server.

42. The process of claim 37, further comprising the step of:

wherein said user-mode client is a copy-on-write user-mode client that allows applications to write configuration or initialization files where they want to without rewriting the application, and without disturbing the local customization of other clients;

wherein each block of data in said persistent cache is marked as clean or dirty;

wherein pages marked as dirty have been customized by the application program and cannot be removed from the cache without losing client customization;

wherein pages marked as clean may be purged from the cache because they can be retrieved again from said server;

providing a cache index;

wherein said cache index indicates which pages in said persistent cache are clean and dirty; and

wherein said copy-on-write user-mode client references said cache index to determine if a page is clean or dirty.

43. The process of claim 37, further comprising the step of:

marking specific files in said persistent cache as not modifiable;

wherein said user-mode client does not allow any data to be written to said specific files that are marked as not modifiable; and

wherein attempts by any processes to mark any of said specific files as modifiable will not succeed.

44. The process of claim 37, further comprising the step of:

maintaining checksums of application code and data in said persistent cache;

wherein when a block of code or data is requested by a local process said streaming file system computes the checksum of the data block before it is returned to the local process; and

wherein if a computed checksum does not match the checksum stored in said persistent cache the cache entry is invalidated and a fresh copy of the page is retrieved from said server.

45. The process of claim 37, further comprising the step of:

assigning each file in an application program a unique identifier;

wherein files that are changed or added in an application upgrade are given new identifiers never before used for that application program;

wherein files that are unchanged retain the same number; and

wherein directories whose contents change are also considered changes. If any file changes, this will cause its parent to change, all the way up to the root directory.

46. The process of claim 45, wherein when an application upgrade occurs said client is given a new root directory for the application program by said server; wherein said new root directory is used by said user-mode client to search for files in the application program; wherein files that do not change can be reused from said persistent cache without downloading them again from said server; and wherein files with new identifiers are retrieved from said server.

47. The process of claim 45, wherein said application upgrades can be marked as mandatory by said server causing the new root directory for the application program to be used immediately; and wherein said user-mode client contacts said server when an application program is started in order to receive any application upgrades.

48. The process of claim 37, wherein said server broadcasts an application program's code and data and any client that is interested in that particular application program stores the broadcasted code and data for later use.

49. An apparatus for client-side retrieval, storage, and execution of application programs and other related data streamed from a server across a computer network to a client system in a computer environment, comprising:

a streaming file system on said client;

wherein said streaming file system appears to said client to contain the installed application program;

wherein said streaming file system receives all requests from local processes for application program code or data that are part of the application;

a persistent cache on said client;

wherein said streaming file system satisfies requests for application program code or data by retrieving it from said persistent cache stored in a native file system or by retrieving it directly from said server; and

wherein application program code or data retrieved from said server is placed in said persistent cache for reuse.

50. The apparatus of claim 49, wherein said persistent cache is encrypted with a key not permanently stored on said client to prevent unauthorized use or duplication of application code or data; and wherein said key is sent to said client upon application startup from said server and said key is not stored in the application program's persistent storage area in said persistent cache.

51. The apparatus of claim 49, wherein said client initiates the prefetching of application program code and data from said server; and wherein said client inspects program code or data file requests and consults the contents of said persistent cache as well as historic information about application program fetching patterns and uses this information to request additional blocks of application program code and data from said server that said client expects will be needed soon.

52. The apparatus of claim 49, wherein said server initiates the prefetching of application program code and data for said client; and wherein said server examines the patterns of requests made by said client and selectively returns to said client additional blocks that said client did not request but is likely to need soon.

53. The apparatus of claim 49, further comprising:

a client-to-client communication mechanism that allows local application customization to migrate from one client machine to another without involving server communication;

wherein when a user wishes to run an application on a second machine, but wishes to retain customizations made previously on the first, said client-to-client mechanism contacts the first machine to retrieve customized files and other customization data; and

wherein unmodified files are retrieved from said server.

54. The apparatus of claim 49, further comprising:

wherein said streaming file system is a copy-on-write file system that allows applications to write configuration or initialization files where they want to without rewriting the application, and without disturbing the local customization of other clients;

wherein each block of data in said persistent cache is marked as clean or dirty;

wherein pages marked as dirty have been customized by the application program and cannot be removed from the cache without losing client customization;

wherein pages marked as clean may be purged from the cache because they can be retrieved again from said server;

a cache index;

wherein said cache index indicates which pages in said persistent cache are clean and dirty; and

wherein said copy-on-write file system references said cache index to determine if a page is clean or dirty.

55. The apparatus of claim 49, further comprising:

a module for marking specific files in said persistent cache as not modifiable;

wherein said streaming file system does not allow any data to be written to said specific files that are marked as not modifiable; and

wherein attempts by any processes to mark any of said specific files as modifiable will not succeed.

56. The apparatus of claim 49, further comprising:

a module for maintaining checksums of application code and data in said persistent cache;

wherein when a block of code or data is requested by a local process said streaming file system computes the checksum of the data block before it is returned to the local process; and

wherein if a computed checksum does not match the checksum stored in said persistent cache the cache entry is invalidated and a fresh copy of the page is retrieved from said server.

57. The apparatus of claim 49, further comprising:

a module for assigning each file in an application program a unique identifier;

wherein files that are changed or added in an application upgrade are given new identifiers never before used for that application program;

wherein files that are unchanged retain the same number; and

wherein directories whose contents change are also considered changes. If any file changes, this will cause its parent to change, all the way up to the root directory.

58. The apparatus of claim 57, wherein when an application upgrade occurs said client is given a new root directory for the application program by said server; wherein said new root directory is used by said streaming file system to search for files in the application program; wherein files that do not change can be reused from said persistent cache without downloading them again from said server; and wherein files with new identifiers are retrieved from said server.

59. The apparatus of claim 57, wherein said application upgrades can be marked as mandatory by said server causing the new root directory for the application program to be used immediately; and wherein said streaming file system contacts said server when an application program is started in order to receive any application upgrades.

60. The apparatus of claim 49, wherein said server broadcasts an application program's code and data and any client that is interested in that particular application program stores the broadcasted code and data for later use.

61. An apparatus for client-side retrieval, storage, and execution of application programs and other related data streamed from a server across a computer network to a client system in a computer environment, comprising:

a kernel-mode streaming file system driver on said client;

a user-mode client on said client;

wherein said streaming file system receives all requests from local processes for application program code or data that are part of the application;

a persistent cache on said client;

wherein requests made to said streaming file system are directed to said user-mode client or retrieved from said persistent cache;

wherein said user-mode client handles the application program code and data streams from said server and sends the results back to said streaming file system driver; and

wherein application program code or data retrieved from said server is placed in said persistent cache for reuse.

62. The apparatus of claim 61, wherein said persistent cache is encrypted with a key not permanently stored on said client to prevent unauthorized use or duplication of application code or data; and wherein said key is sent to said client upon application startup from said server and said key is not stored in the application program's persistent storage area in said persistent cache.

63. The apparatus of claim 61, wherein said client initiates the prefetching of application program code and data from said server; and wherein said client inspects program code or data file requests and consults the contents of said persistent cache as well as historic information about application program fetching patterns and uses this information to request additional blocks of application program code and data from said server that said client expects will be needed soon.

64. The apparatus of claim 61, wherein said server initiates the prefetching of application program code and data for said client; and wherein said server examines the patterns of requests made by said client and selectively returns to said client additional blocks that said client did not request but is likely to need soon.

65. The apparatus of claim 61, further comprising:

a client-to-client communication mechanism that allows local application customization to migrate from one client machine to another without involving server communication;

wherein when a user wishes to run an application on a second machine, but wishes to retain customizations made previously on the first, said client-to-client mechanism contacts the first machine to retrieve customized files and other customization data; and

wherein unmodified files are retrieved from said server.

66. The apparatus of claim 61, further comprising:

wherein said streaming file system is a copy-on-write file system that allows applications to write configuration or initialization files where they want to without rewriting the application, and without disturbing the local customization of other clients;

wherein each block of data in said persistent cache is marked as clean or dirty;

wherein pages marked as dirty have been customized by the application program and cannot be removed from the cache without losing client customization;

wherein pages marked as clean may be purged from the cache because they can be retrieved again from said server;

a cache index;

wherein said cache index indicates which pages in said persistent cache are clean and dirty; and

wherein said copy-on-write file system references said cache index to determine if a page is clean or dirty.

67. The apparatus of claim 61, further comprising:

a module for marking specific files in said persistent cache as not modifiable;

wherein said streaming file system does not allow any data to be written to said specific files that are marked as not modifiable; and

wherein attempts by any processes to mark any of said specific files as modifiable will not succeed.

68. The apparatus of claim 61, further comprising:

a module for maintaining checksums of application code and data in said persistent cache;

wherein when a block of code or data is requested by a local process said streaming file system computes the checksum of the data block before it is returned to the local process; and

wherein if a computed checksum does not match the checksum stored in said persistent cache the cache entry is invalidated and a fresh copy of the page is retrieved from said server.

69. The apparatus of claim 61, further comprising:

a module for assigning each file in an application program a unique identifier;

wherein files that are changed or added in an application upgrade are given new identifiers never before used for that application program;

wherein files that are unchanged retain the same number; and

wherein directories whose contents change are also considered changes. If any file changes, this will cause its parent to change, all the way up to the root directory.

70. The apparatus of claim 69, wherein when an application upgrade occurs said client is given a new root directory for the application program by said server; wherein said new root directory is used by said streaming file system to search for files in the application program; wherein files that do not change can be reused from said persistent cache without downloading them again from said server; and wherein files with new identifiers are retrieved from said server.

71. The apparatus of claim 69, wherein said application upgrades can be marked as mandatory by said server causing the new root directory for the application program to be used immediately; and wherein said streaming file system contacts said server when an application program is started in order to receive any application upgrades.

72. The apparatus of claim 61, wherein said server broadcasts an application program's code and data and any client that is interested in that particular application program stores the broadcasted code and data for later use.

73. An apparatus for client-side retrieval, storage, and execution of application programs and other related data streamed from a server across a computer network to a client system in a computer environment, comprising:

a streaming block driver on said client;

wherein said block driver provides the abstraction of a physical disk to a native file system already installed on the client operating system;

a persistent cache on said client;

wherein said block driver receives requests for physical block reads and writes from local processes which it satisfies out of said persistent cache on a standard file system that is backed by a physical disk drive; and

wherein requests that cannot be satisfied by said persistent cache are sent to said server.

74. The apparatus of claim 73, wherein said persistent cache is encrypted with a key not permanently stored on said client to prevent unauthorized use or duplication of application code or data; and wherein said key is sent to said client upon application startup from said server and said key is not stored in the application program's persistent storage area in said persistent cache.

75. The apparatus of claim 73, wherein said client initiates the prefetching of application program code and data from said server; and wherein said client inspects program code or data file requests and consults the contents of said persistent cache as well as historic information about application program fetching patterns and uses this information to request additional blocks of application program code and data from said server that said client expects will be needed soon.

76. The apparatus of claim 73, wherein said server initiates the prefetching of application program code and data for said client; and wherein said server examines the patterns of requests made by said client and selectively returns to said client additional blocks that said client did not request but is likely to need soon.

77. The apparatus of claim 73, further comprising:

a client-to-client communication mechanism that allows local application customization to migrate from one client machine to another without involving server communication;

wherein when a user wishes to run an application on a second machine, but wishes to retain customizations made previously on the first, said client-to-client mechanism contacts the first machine to retrieve customized files and other customization data; and

wherein unmodified files are retrieved from said server.

78. The apparatus of claim 73, further comprising:

wherein said block driver is a copy-on-write block driver that allows applications to write configuration or initialization files where they want to without rewriting the application, and without disturbing the local customization of other clients;

wherein each block of data in said persistent cache is marked as clean or dirty;

wherein pages marked as dirty have been customized by the application program and cannot be removed from the cache without losing client customization;

wherein pages marked as clean may be purged from the cache because they can be retrieved again from said server;

a cache index;

wherein said cache index indicates which pages in said persistent cache are clean and dirty; and

wherein said copy-on-write block driver references said cache index to determine if a page is clean or dirty.

79. The apparatus of claim 73, further comprising:

a module for marking specific files in said persistent cache as not modifiable;

wherein said block driver does not allow any data to be written to said specific files that are marked as not modifiable; and

wherein attempts by any processes to mark any of said specific files as modifiable will not succeed.

80. The apparatus of claim 73, further comprising:

a module for maintaining checksums of application code and data in said persistent cache;

wherein when a block of code or data is requested by a local process said streaming file system computes the checksum of the data block before it is returned to the local process; and

wherein if a computed checksum does not match the checksum stored in said persistent cache the cache entry is invalidated and a fresh copy of the page is retrieved from said server.

81. The apparatus of claim 73, further comprising:

a module for assigning each file in an application program a unique identifier;

wherein files that are changed or added in an application upgrade are given new identifiers never before used for that application program;

wherein files that are unchanged retain the same number; and

wherein directories whose contents change are also considered changes. If any file changes, this will cause its parent to change, all the way up to the root directory.

82. The apparatus of claim 81, wherein when an application upgrade occurs said client is given a new root directory for the application program by said server; wherein said new root directory is used by said block driver to search for files in the application program; wherein files that do not change can be reused from said persistent cache without downloading them again from said server; and wherein files with new identifiers are retrieved from said server.

83. The apparatus of claim 81, wherein said application upgrades can be marked as mandatory by said server causing the new root directory for the application program to be used immediately; and wherein said block driver contacts said server when an application program is started in order to receive any application upgrades.

84. The apparatus of claim 73, wherein said server broadcasts an application program's code and data and any client that is interested in that particular application program stores the broadcasted code and data for later use.

85. An apparatus for client-side retrieval, storage, and execution of application programs and other related data streamed from a server across a computer network to a client system in a computer environment, comprising:

a disk driver on said client;

a user mode client on said client;

wherein said disk driver sends all file requests that it receives to said user-mode client;

a persistent cache on said client; and

wherein said user-mode client attempts to satisfy said file requests from said program cache or by making requests from said server.

86. The apparatus of claim 85, wherein said persistent cache is encrypted with a key not permanently stored on said client to prevent unauthorized use or duplication of application code or data; and wherein said key is sent to said client upon application startup from said server and said key is not stored in the application program's persistent storage area in said persistent cache.

87. The apparatus of claim 85, wherein said client initiates the prefetching of application program code and data from said server; and wherein said client inspects program code or data file requests and consults the contents of said persistent cache as well as historic information about application program fetching patterns and uses this information to request additional blocks of application program code and data from said server that said client expects will be needed soon.

88. The apparatus of claim 85, wherein said server initiates the prefetching of application program code and data for said client; and wherein said server examines the patterns of requests made by said client and selectively returns to said client additional blocks that said client did not request but is likely to need soon.

89. The apparatus of claim 85, further comprising:

a client-to-client communication mechanism that allows local application customization to migrate from one client machine to another without involving server communication;

wherein when a user wishes to run an application on a second machine, but wishes to retain customizations made previously on the first, said client-to-client mechanism contacts the first machine to retrieve customized files and other customization data; and

wherein unmodified files are retrieved from said server.

90. The apparatus of claim 85, further comprising:

wherein said user-mode client is a copy-on-write user-mode client that allows applications to write configuration or initialization files where they want to without rewriting the application, and without disturbing the local customization of other clients;

wherein each block of data in said persistent cache is marked as clean or dirty;

wherein pages marked as dirty have been customized by the application program and cannot be removed from the cache without losing client customization;

wherein pages marked as clean may be purged from the cache because they can be retrieved again from said server;

a cache index;

wherein said cache index indicates which pages in said persistent cache are clean and dirty; and

wherein said copy-on-write user-mode client references said cache index to determine if a page is clean or dirty.

91. The apparatus of claim 85, further comprising:

a module for marking specific files in said persistent cache as not modifiable;

wherein said user-mode client does not allow any data to be written to said specific files that are marked as not modifiable; and

wherein attempts by any processes to mark any of said specific files as modifiable will not succeed.

92. The apparatus of claim 85, further comprising:

a module for maintaining checksums of application code and data in said persistent cache;

wherein when a block of code or data is requested by a local process said streaming file system computes the checksum of the data block before it is returned to the local process; and

wherein if a computed checksum does not match the checksum stored in said persistent cache the cache entry is invalidated and a fresh copy of the page is retrieved from said server.

93. The apparatus of claim 85, further comprising:

a module for assigning each file in an application program a unique identifier;

wherein files that are changed or added in an application upgrade are given new identifiers never before used for that application program;

wherein files that are unchanged retain the same number; and

wherein directories whose contents change are also considered changes. If any file changes, this will cause its parent to change, all the way up to the root directory.

94. The apparatus of claim **93**, wherein when an application upgrade occurs said client is given a new root directory for the application program by said server; wherein said new root directory is used by said user-mode client to search for files in the application program; wherein files that do not change can be reused from said persistent cache without downloading them again from said server; and wherein files with new identifiers are retrieved from said server.

95. The apparatus of claim **93**, wherein said application upgrades can be marked as mandatory by said server causing the new root directory for the application program to be used immediately; and wherein said user-mode client contacts said server when an application program is started in order to receive any application upgrades.

96. The apparatus of claim 85, wherein said server broadcasts an application program's code and data and any client that is interested in that particular application program stores the broadcasted code and data for later use.

* * * * *

US006412009B1

(12) United States Patent

Erickson et al.

(10) Patent No.: **US 6,412,009 B1**

(45) Date of Patent: **Jun. 25, 2002**

(54) **METHOD AND SYSTEM FOR PROVIDING A PERSISTENT HTTP TUNNEL**

(75) Inventors: **Rodger D. Erickson**, St. Louis, MO (US); **Ronald D. Sanders**, Spokane, WA (US)

(73) Assignee: **Wall Data Incorporated**, Kirkland, WA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/268,068**

(22) Filed: **Mar. 15, 1999**

(51) **Int. Cl.**^{7} **G06F 13/00**

(52) **U.S. Cl.** **709/228**; 709/217; 709/236; 709/313

(58) **Field of Search** 709/203, 217, 709/219, 223, 224, 225, 227, 228, 236, 238, 313

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,754,830 A	*	5/1998	Butts et al.	395/500.44
5,778,372 A	*	7/1998	Cordell et al.	707/100
5,935,212 A	*	8/1999	Kalajan et al.	709/228
5,941,988 A	*	8/1999	Bhagwat et al.	713/201

OTHER PUBLICATIONS

Provan, D., "Tunneling IPX Traffic through IP Networks," RFC 1234, Novell, Inc. (Jun. 1991).

Woodburn, R. et al., "A Scheme for an Internet Encapsulation Protocol: Version 1," RFC 1241, University of Delaware (Jul. 1991).

* cited by examiner

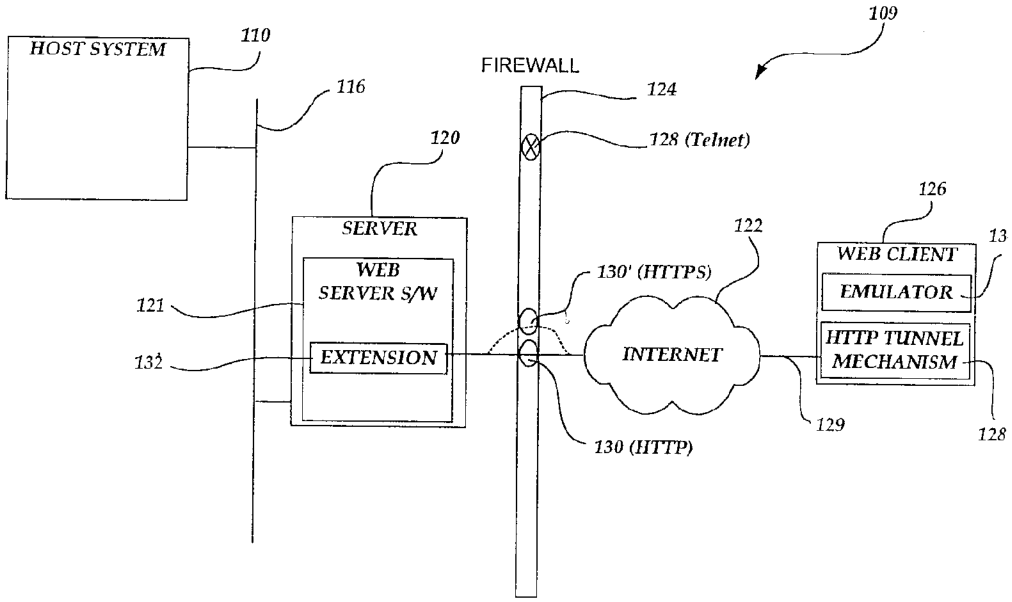
Primary Examiner—Viet D. Vu

(74) *Attorney, Agent, or Firm*—Christensen O'Oconnor Johnson Kindness PLLC

(57) **ABSTRACT**

A method and system for providing a persistent HTTP tunnel for a connection-oriented protocol between a client and a Web server. A data message complying with the connection-oriented protocol is generated and embedded into a chunked data message in compliance with a chunking option for the HTTP. The chunked data message is transmitted between a client and a Web server. Upon receiving any chunked data message at the Web server, the Web server parses the chunked data message and delivers the data message to a host system. Upon receiving any chunked data message at the client, the client parses the chunked data message and delivers the data message to a terminal emulator running on the client. This allows a terminal session to be supported by a real-time bi-directional persistent connection with the host system. The bi-directional persistent connection allows interleaving of the chunked data messages from the Web client with the chunked data messages on the Web server on the persistent HTTP tunnel.

8 Claims, 5 Drawing Sheets



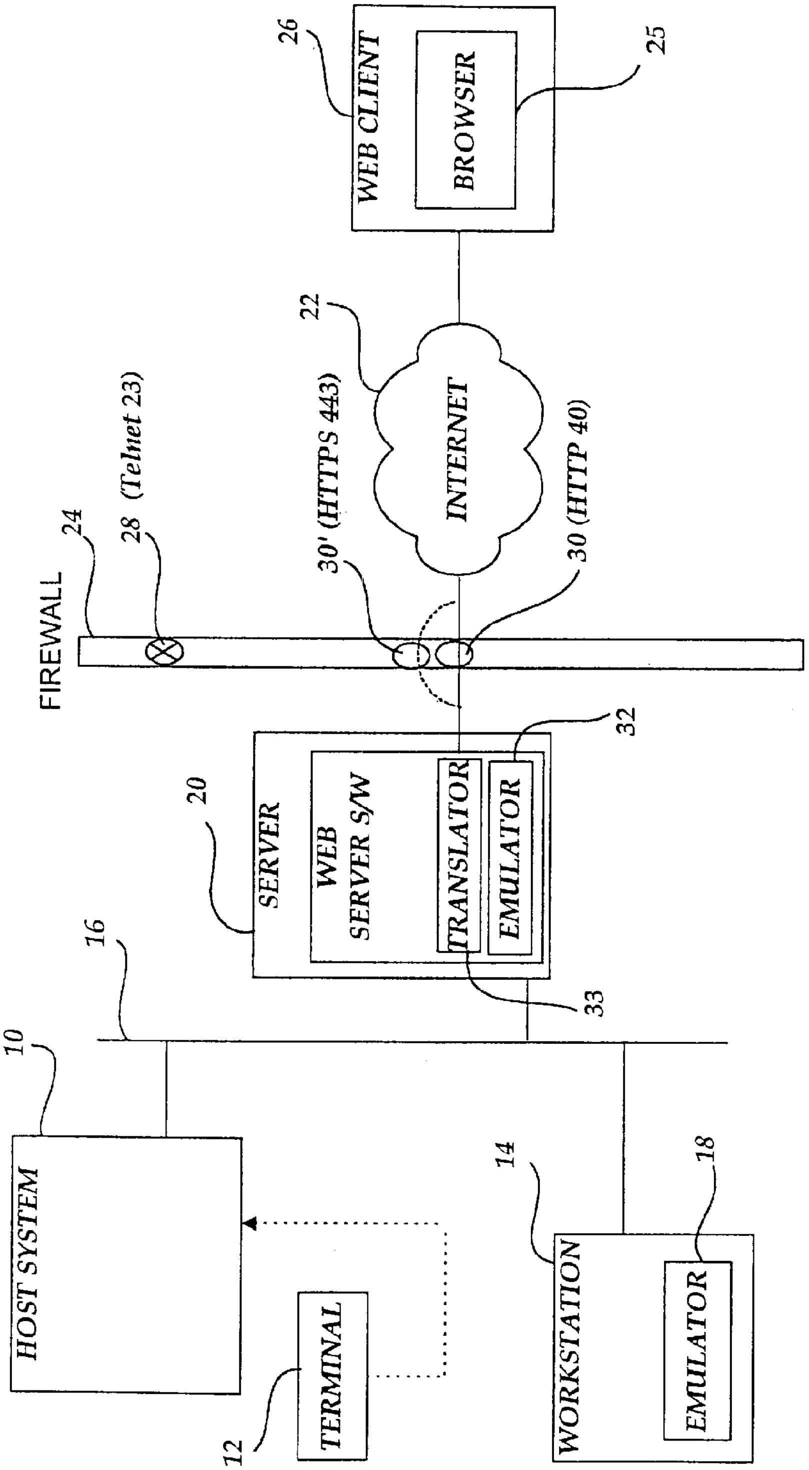


Fig.1.
PRIOR ART

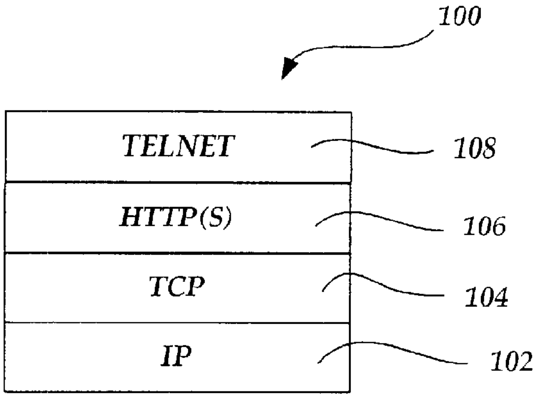
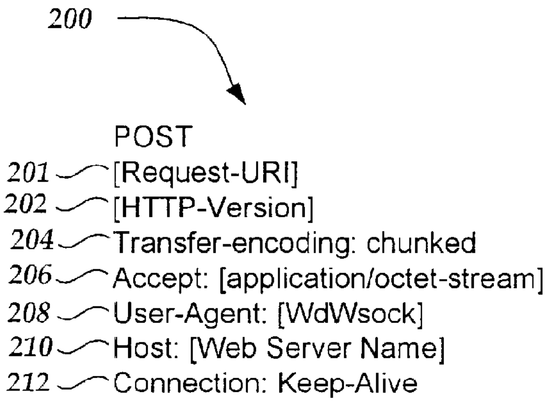
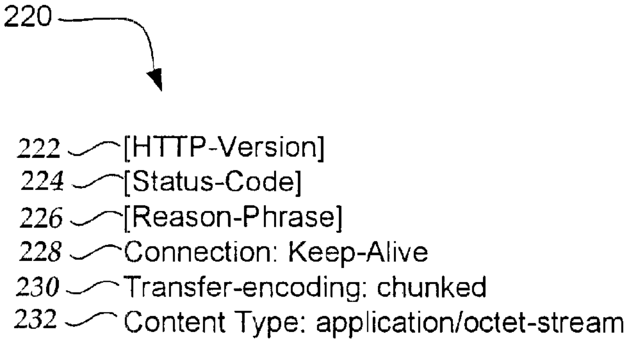


Fig.2.



POST
[Request-URI]
[HTTP-Version]
Transfer-encoding: chunked
Accept: [application/octet-stream]
User-Agent: [WdWsock]
Host: [Web Server Name]
Connection: Keep-Alive

Fig.5.



[HTTP-Version]
[Status-Code]
[Reason-Phrase]
Connection: Keep-Alive
Transfer-encoding: chunked
Content Type: application/octet-stream

Fig.6.

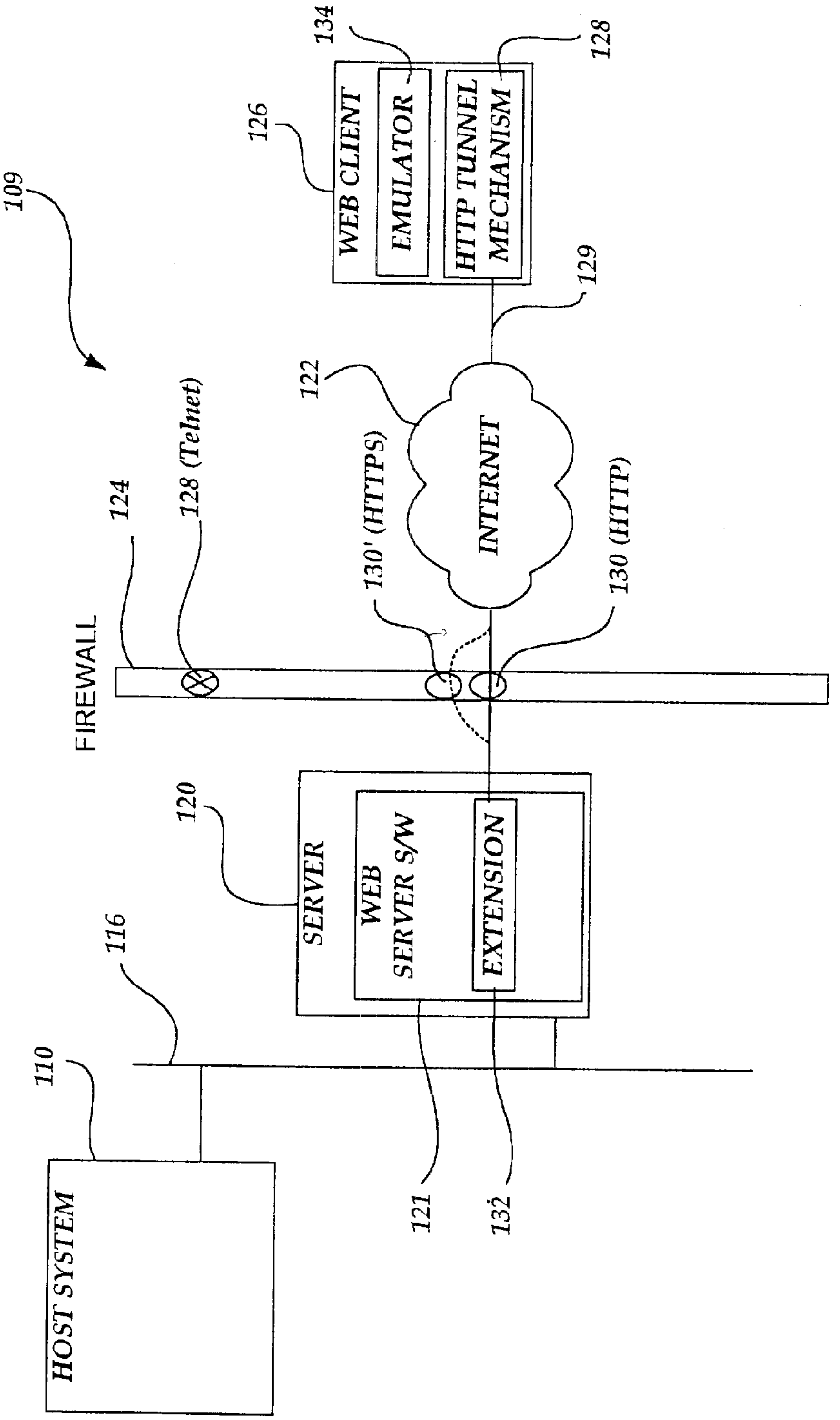

109
WEB CLIENT
126
EMULATOR
134
HTTP TUNNEL MECHANISM
128
129
INTERNET
122
130' (HTTPS)
130 (HTTP)
FIREWALL
124
128 (Telnet)
SERVER
120
WEB SERVER S/W
EXTENSION
116
121
132
HOST SYSTEM
110

Fig.3.

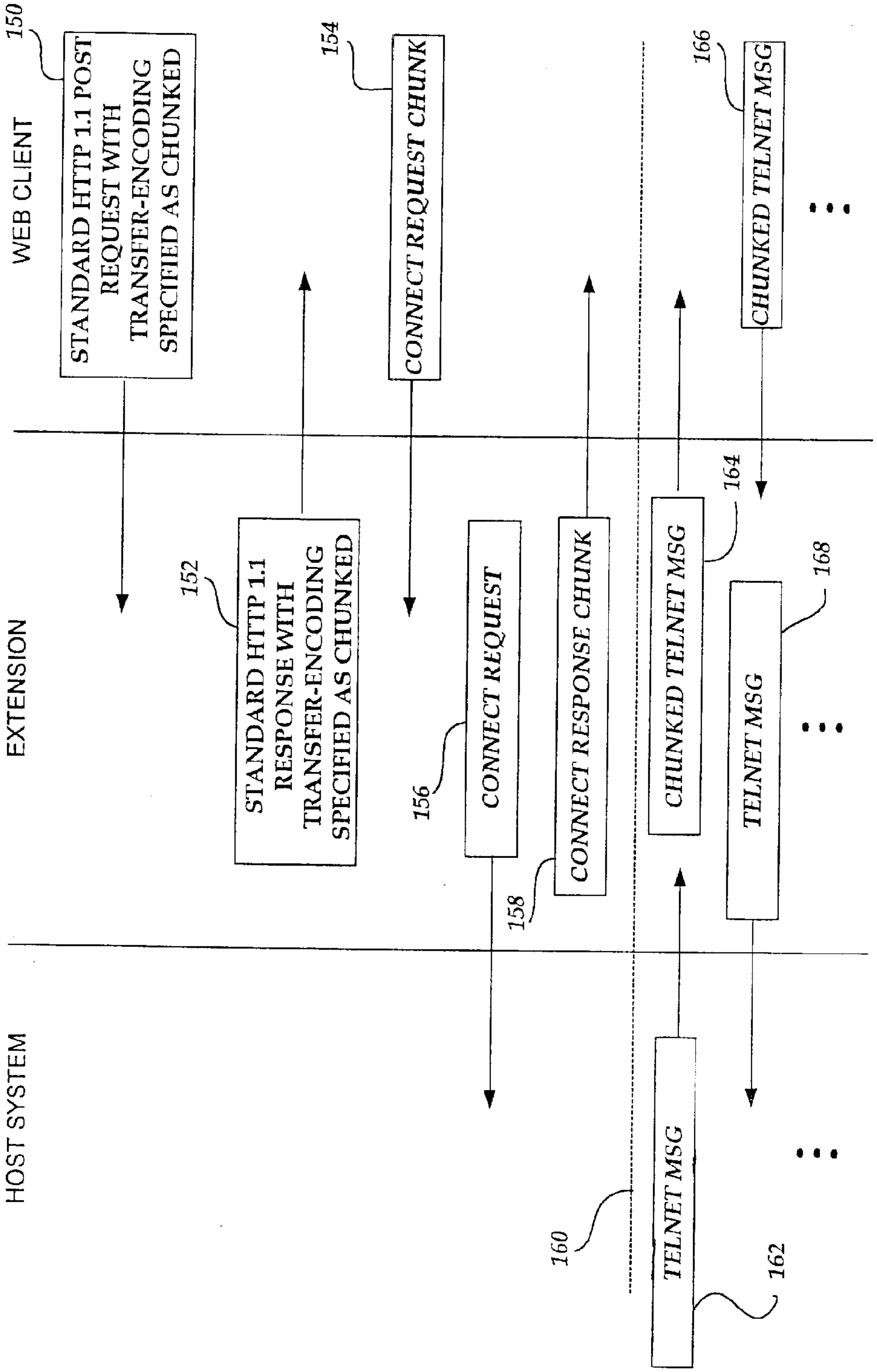

WEB CLIENT
EXTENSION
HOST SYSTEM
150
STANDARD HTTP 1.1 POST REQUEST WITH TRANSFER-ENCODING SPECIFIED AS CHUNKED
152
STANDARD HTTP 1.1 RESPONSE WITH TRANSFER-ENCODING SPECIFIED AS CHUNKED
154
CONNECT REQUEST CHUNK
156
CONNECT REQUEST
158
CONNECT RESPONSE CHUNK
160
162
TELNET MSG
164
CHUNKED TELNET MSG
166
CHUNKED TELNET MSG
168
TELNET MSG
...
...
...

Fig.4.

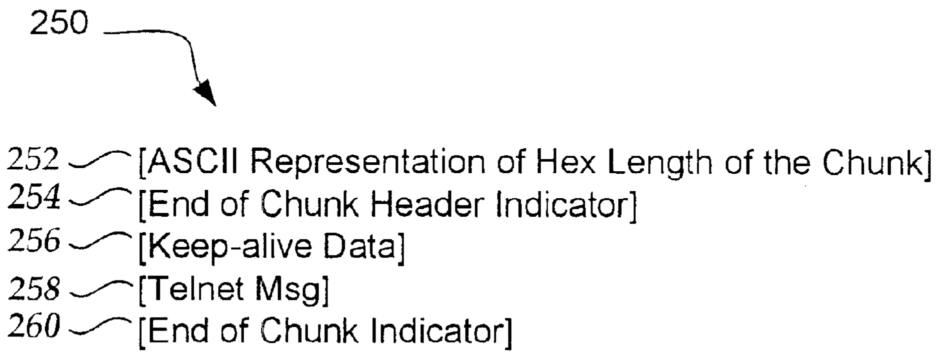

250
252 [ASCII Representation of Hex Length of the Chunk]
254 [End of Chunk Header Indicator]
256 [Keep-alive Data]
258 [Telnet Msg]
260 [End of Chunk Indicator]

Fig.7.

PRIOR ART

METHOD AND SYSTEM FOR PROVIDING A PERSISTENT HTTP TUNNEL

FIELD OF THE INVENTION

The present invention relates generally to communications between two computers, and, more particularly, to a communication method and system that provides a persistent HTTP tunnel for a connection-oriented protocol between the two computers.

BACKGROUND OF THE INVENTION

Many corporations continue to maintain their corporate computer data on what are referred to as "legacy host" systems. These systems are generally older mainframe or mini-computers which cannot be easily replaced. As personal computers become more commonplace, a significant effort has been devoted to methods by which a user of a personal computer can access data stored on a legacy host. One such system is shown in FIG. **1**, whereby access to a host system **10** may be through a terminal **12** directly coupled to the host system **10** that provides a local login capability. If a workstation **14** desires to connect to the host system **10** over a Local Area Network (LAN) **16**, the workstation **14** runs a terminal emulator **18**. A Telnet protocol is used between the host system **10** and the workstation **14** to provide the workstation **14** access to the host system **10** as if the workstation was a local terminal directly connected to the host system.

With the phenomenal growth of the Internet, a need developed to provide the ability of a user to access a host computer from anywhere in the world. A typical configuration to allow such Internet access includes a Web server **20** coupled to the host system **10** via the LAN **16**. The Web server **20** is then coupled to the Internet **22** through a firewall **24**. The firewall **24** enforces a security policy between a secure internal network containing the host system **10** and an untrusted network such as the Internet. The firewall may be a personal computer (PC), a router, a mainframe, a UNIX workstation, or any combination of such devices responsible for enforcing the security policy. A Web client computer **26** runs a browser program **25** to access the Web server **20** through the Internet **22**.

Because the Internet **22** is an unsecured network, most firewall security policies will not allow the Web client **26** to communicate using the well-known "Telnet port 23," shown at **28**. However, most firewalls allow communications through the well-known "HyperText Transfer Protocol (HTTP) port 80," shown at **30**, and the secure "HTTP port 443," shown at **30'**. Therefore, a current system that provides a local login experience to a Web client **26** uses one of these HTTP ports **30 30'**. In this system, the Web server **20** runs a terminal emulator **32** that provides a Telnet session with the host system **10**. The Web server **20** receives the Telnet data from the host system **10** and instead of displaying the data as a typical text screen will instead send the Telnet data to a translator **33**. The translator **33** translates the Telnet data to HyperText Markup Language (HTML) statements that are sent to the browser program **25** running on the Web client **26**. The browser **25** then translates the HTML statements and displays an HTML page on the Web client.

A problem with this type of system is that the translated HTML screen does not look sufficiently similar to a local login screen that the user would see if they were directly connected to the host system and the interaction with the HTML screen is not sufficiently similar to the interaction with the directly connected terminal. In certain situations, the differences may require additional training of the users on the Web client. Another problem is that the response times between a user request on the Web client **26** and the return response from the host system is more variable in comparison with a local login response time. For example, response times may range from a second to thirty seconds using the HTML screen, in comparison with response times in the range of one second to three seconds with a directly connected terminal. The variable response times are due to the nature of the HTTP protocol.

HTTP is a request/response protocol. The Web client **26**, using the browser **25**, establishes a connection with the Web server **20** and sends a request to the Web server. After the Web server sends a response to the browser **25**, the connection is closed. Before additional requests may be handled, a new connection must be established. Even though the newer HTTP 1.1 specification provides a keep-alive mechanism that allows one connection for multiple objects on an HTML page, the connection is closed either by the Web server or the browser after a period of inactivity. The period of inactivity may range from several seconds to a fraction of a second depending on the activity on the Web server. Many protocols, including Telnet, have insufficient transaction rates to maintain an alive connection even when the Web server is only modestly loaded. Closing the connection and establishing a new connection creates significant overhead resulting in decreased performance.

Given the shortcomings associated with the prior art method of providing access to host computer systems for Web clients, there is a need for a method that uses existing standard ports in the firewall while providing more consistent response times similar to the response times of a workstation connected through a LAN to a host system. The present invention is directed to filling this need.

SUMMARY OF THE INVENTION

In accordance with this invention, a server, a client, and a method of operation are provided for a Web client to access a host system with performance and displays comparable to the performance and displays of a workstation connected through a LAN to the host system.

In accordance with one aspect of this invention, a method of providing a persistent HTTP tunnel for a persistent virtual session is provided. A data message complying with a connection-oriented protocol is generated at an endpoint of a connection-oriented virtual session. The data message is embedded into a chunked data message complying with a chunking option of an HTTP specification. The chunked data message is transmitted between a Web client and a Web server via an HTTP connection. Upon receiving any chunked data message at the Web server, the Web server parses the chunked data message and delivers the data message to one endpoint of the connection-oriented virtual session. In the other direction, upon receiving any chunked data message at the Web client, the Web client parses the chunked data message and delivers the data message to another endpoint of the connection-oriented virtual session. The chunked data messages from the Web client are interleaved with the chunked data messages from the Web server on the persistent HTTP tunnel.

In accordance with other aspects of this invention, the connection-oriented protocol is a Telnet protocol.

In accordance with still further aspects of this invention, one endpoint of the connection-oriented virtual session is a host system.

In accordance with yet further aspects of this invention, the other endpoint of the connection-oriented session is a Web client application.

In accordance with still other aspects of this invention, the Web client application is a terminal emulator.

In accordance with another aspect of this invention, a method for creating a persistent tunnel between a Web client and a Web server using an HTTP protocol for providing a persistent virtual connection between a host system and the Web client is provided. A connection between the Web client and the Web server is established using a chunking option in accordance with an HTTP protocol that allows a series of messages to be sent as chunked messages. A virtual session is established between the host system and the Web client through a Web Server extension. A plurality of host messages are transmitted from the host system to the Web server extension and inserted into a chunked host message at the Web server. The Web server forwards the chunked host messages to the Web client over the connection. The Web client parses the chunked host message and delivers the host message to an application. In the other direction, a plurality of Web client messages are transmitted from the application to a tunneling mechanism on the Web client. The tunneling mechanism inserts the client message into a chunked client message and forwards the chunked client message to the Web server over the connection. The Web server forwards the chunked client message to an extension before receiving subsequent chunked client messages. The extension parses the chunked client message and delivers the client message to the host system. The chunked data messages from the Web client are interleaved with the chunked data messages from the Web server on the persistent HTTP tunnel.

In accordance with a further aspect of the present invention, a server for providing a persistent virtual session over HTTP is provided. The server includes a server software component operable to communicate via a persistent HTTP tunnel with a first endpoint of a connection-oriented session. The server also includes an extension operable to communicate with the server software component and a second endpoint of a connection-oriented session. Upon a connect request from a client, the extension establishes a connection-oriented session with the second endpoint to provide a virtual connection-oriented session between the first endpoint and the second endpoint. After the virtual connection-oriented session is established, the extension receives one or more chunked client messages from the client. The chunked messages comply with a chunking option as specified in the Hyper Text Transfer Protocol. Each chunked client message includes a chunk header and a data portion. The extension forwards the data portion to the second endpoint over the connection-oriented session. The extension also receives one or more second endpoint messages from the second endpoint and encapsulates each second endpoint message into a chunked second endpoint message. The extension then forwards the chunked second endpoint message to the client that delivers the second endpoint message of the chunked second endpoint message to the first endpoint.

In accordance with other aspects of this invention, the second endpoint is a host system.

In accordance with still further aspects of this invention, the first endpoint is a client application.

In accordance with yet still further aspects of this invention, the client application is a terminal emulator.

In accordance with a further aspect of this invention, a client having a first endpoint of a connection-oriented session having a persistent virtual session with a second endpoint over HTTP is provided. The client includes an application for sending and receiving data messages complying with the connection-oriented session at the second endpoint and an HTTP tunnel mechanism. The HTTP tunnel mechanism receives the data messages generated by the application and inserts the data messages into a chunked data message complying with HTTP and transmits the chunked data messages to a Web server. The HTTP tunnel mechanism also receives chunked data messages generated by the Web server and forwards the data messages within the chunked data message to the application.

A technical advantage of the present invention is the ability to establish a connection-oriented virtual session between a host system and a Web client through the commonly available HTTP port. Both the Web server and the Web client encapsulate the connection-oriented session data, such as Telnet, into chunks that comply with the HTTP specification. The Web server transmits the HTTP response as soon as the request is received and transmits chunked messages and receives chunked messages from the Web client in an interleaving manner without completing the original request. Because the chunks are interleaved, the present invention provides a persistent bi-directional virtual connection between the host system and the Web client.

BRIEF DESCRIPTION OF THE DRAWINGS

The foregoing aspects and many of the attendant advantages of this invention will become more readily appreciated as the same become better understood by reference to the following detailed description, when taken in conjunction with the accompanying drawings, wherein:

FIG. **1** is a block diagram of a prior art method for connecting Web clients to host computer systems;

FIG. **2** is a block diagram of a protocol stack for tunneling Telnet data on HTTP;

FIG. **3** is a block diagram of a system for connecting Web clients to host systems in accordance with the present invention;

FIG. **4** is a block diagram illustrating a communication flow among a Web client, an extension residing on a Web server, and a host system in accordance with the present invention;

FIG. **5** is a format for an HTTP Post request suitable for use in the present invention;

FIG. **6** is a format for an HTTP response message to the HTTP Post request of FIG. **5** suitable for use in the present invention; and

FIG. **7** is a format of an HTTP chunk message suitable for use in the present invention.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

In accordance with the present invention, access to host systems from Web clients is available in a manner that provides performance and displays comparable to the performance and displays of a workstation connected through a LAN to the host system. The present invention provides a system and method for providing a persistent HTTP tunnel for any connection-oriented protocol and keeping the HTTP tunnel connection between the Web client and the Web server persistent for the duration of the communication between the Web client and host system. By providing a persistent connection for the entire duration, the present invention achieves performance comparable to the performance of the workstation **14** connected through the LAN **16** to the host system, as shown in FIG. **1**. In addition, the present invention utilizes an existing port in the firewall rather than requiring a new hole in the firewall.

Although the presently preferred embodiment of the invention utilizes the connection-oriented protocol Telnet, those skilled in the art will recognize that other connection-oriented protocols may be used. As mentioned earlier, the Telnet protocol is typically used between a host system and a client residing on a network to provide the client access to the host system as if the client was a local terminal directly connected to the host system.

FIG. **2** is a block diagram illustrating how Telnet data is tunneled through HTTP in accordance with the present invention. A protocol stack **100** of the present invention has a first or lower layer **102** that represents an Internet layer or a network layer that shields higher layers **104**–**108** from a physical network architecture. An Internet Protocol (IP) is the protocol of the first layer **102** and is a connectionless protocol that does not provide reliability, flow control or error recovery. The IP provides a routing function that ensures that messages will be correctly delivered to their destination. A second layer **104** is a transport layer that provides an end-to-end reliable data transfer. The second layer **104** allows multiple applications to be supported simultaneously. A Transmission Control Protocol (TCP) is used in the second layer **104** to provide a reliable exchange of information. A third layer **106** is an application layer. An interface between the application layer and the transport layer is defined by port numbers and sockets. In contrast to prior art methods of implementing Telnet on top of TCP, the present invention tunnels Telnet on top of the HTTP(S) protocol of the third layer **106** resulting in a fourth layer **108** of the protocol stack **100**. One skilled in the art will appreciate that HTTP is considered the "official" application protocol of the World Wide Web and Telnet is considered the "official" protocol for emulating a remote terminal. The tunneling of the Telnet protocol on top of the HTTP protocol will be described in greater detail below.

FIG. **3** is a block diagram of a system for connecting Web clients to host systems in accordance with the present invention. A system **109** includes a Web server **120** coupled to a host system **110** via a LAN **116**. The Web server **120** is also coupled to an Internet **122** through a firewall **124**. The firewall **124** enforces a security policy between a secure internal network containing the host system **110** and an untrusted network such as the Internet. The firewall **124** may be a personal computer (PC), a router, a mainframe, a UNIX workstation, or any combination of such devices responsible for enforcing the security policy.

A Web client **126** is coupled to the Web server **120** through the Internet **122**. The system **109** further comprises an extension **132** residing on the Web server **120**. The extension **132** may run in the same process space as Web server software **121** or communicate with the Web server software **121** using an interprocess communication well-known in the art. The Web server software **121** passes all commands and data designating the extension **132** to the extension **132**. The commands and data are received from the Web client through a HyperText Transfer Protocol (HTTP) port **80**, shown at **130**, or the secure HTTP port **443**, shown at **130**' in the firewall **124**. As one skilled in the art will appreciate, the firewall **124** passes the commands and data from the Web client onto the Web server **120**. Because the present invention utilizes the standard HTTP ports **130** and **130**', additional holes in the firewall are not necessary. In addition, because the present invention utilizes the standard HTTP ports, the present invention minimizes the amount of development necessary to create a secure Web server environment. Rather, the present invention can utilize existing resources residing on the Web server, thereby leveraging their development costs. The extension **132** is responsible for providing a virtual session between the host system **110** and the Web client **126**.

The Web client **126** comprises an HTTP tunnel mechanism **128** and an emulator **134**. The HTTP tunnel mechanism **128** is a software module that provides an Application Programming Interface (API) for TCP, such as Winsock created by Microsoft Corporation, of Redmond, Wash., or WDWSock created by Wall Data Incorporated, of Kirkland, Wash. The HTTP tunnel mechanism **128** receives messages created in the extension **132**, parses the messages, and passes any Telnet data or commands to the emulator **134** so that the emulator **134** can emulate a host terminal.

The present invention utilizes a chunking option as specified in the HyperText Transfer Protocol specification 1.1 dated January 1997, referred to as HTTP/1.1 (RFC 2068), the specification of which is hereby incorporated by reference. The chunking option refers to a new transfer coding implementation that allows a body of a message to be transferred as a series of chunks, "chunked data," each with its own size indicator, followed by an optional footer containing entity-header fields. Prior to the chunking option, the size of the message along with the entire message must be sent at the same time. The present invention expands the use of the chunking option by sending a series of HTTP messages with embedded session-oriented data, "chunked messages," between the Web server and the Web client without sending an end chunk message. This allows one connection to remain active during the series of interleaved HTTP messages between the Web server and the Web client, thereby creating an HTTP tunnel **129** that is persistent for the duration of the communication between the host system and the Web client. The interleaved HTTP messages include messages that alternate between the Web client and the Web server as the sender for each message or messages that alternate between one or more Web client messages and then one or more Web server messages. Therefore, a persistent virtual connection between two endpoints of the connection-oriented protocol is provided. One endpoint located on the host system and the other endpoint on the Web client. In the embodiment described, a terminal emulator is the endpoint on the Web client and a Telnet process is the endpoint on the host system.

Depending upon the type of Web server **120**, a setting may need to be changed to inform the Web server **120** not to process chunked data designating the extension **132**. By changing the setting, the extension is allowed to handle the chunked data in a manner described in detail below. To change the setting of any Web server **120** running Microsoft Internet Information Server (IIS) 4.0 by Microsoft Corporation, the extension **132** accesses a metabase and programmatically changes an MD_UPLOAD_READAHEAD_SIZE parameter to zero (0) for the named extension **132**. The MD_UPLOAD_READAHEAD_SIZE parameter is typically used to fine tune memory usage on the Web server **120**. Typically, the MD_UPLOAD_READAHEAD SIZE parameter is set to a value that is an average size of a request message received from a Web client **126**. The present invention therefore modifies the MD_UPLOAD_READAHEAD_SIZE parameter for an unusual purpose of having the Web server software **121** ignore processing of chunked data that is designated to be passed to the extension **132**. By disabling the processing of chunked data in the Web server software **121**, the extension **132** can provide the processing for the chunked data, thereby allowing a connection-oriented protocol to be tunneled on top of HTTP and creating a bi-directional connection that remains active for the duration of the communication between a Web client and a host system.

FIG. **4** is a block diagram illustrating the communication flow among the Web client **126**, the extension **132**, and the host system **110**. Again, Telnet is used as the connection-oriented protocol and is given by way of nonlimiting example only. At block **150**, the Web client **126** generates a standard HTTP/1.1 post request with transfer-encoding specified as chunked. The creation of the post request will be described in greater detail below. At block **152**, the extension receives the post request, and generates a standard HTTP/1.1 response with transfer-encoding specified as chunked which is sent back to the Web client. At block **154**, the Web client generates and sends a connect request chunk to the extension. The connection request chunk includes a destination port field for specifying the port number of the host system for connection. In the above-described embodiment, the destination port field contains a value specifying the Telnet port **23**. The connect request chunk also includes a destination IP address field for specifying the HP address for the host system. Because the connect request chunk occurs after the initial post response, the connect request chunk uses the format for chunked data. At block **156**, the extension sends a connect request to the host system to establish a Telnet session. At block **158**, the extension generates and sends a connect response chunk to the Web client. After the Web client receives the connect response chunk, the end-to-end session between the Web client and the host system via the extension is active.

The communication flow occurring after the reference line **160** represents Telnet data tunneled over HTTP. A typical exchange has the host system starting Telnet negotiation. At block **162**, the host system generates a Telnet message. The Telnet message includes Telnet commands and/or Telnet data and is sent to the extension. At block **164**, the extension encapsulates the Telnet message into a chunked Telnet message by embedding the Telnet message in a chunk that complies with the HTTP/1.1 format with Transfer-Encoding specified as chunked. The chunked Telnet message is sent via the HTTP tunnel **129** through the HTTP port over the Internet to the Web client.

The HTTP tunnel mechanism **128** in the Web client **126** parses the chunked Telnet message and provides the emulator **134** the original Telnet message as sent from the host system. Whenever the emulator generates a Telnet message, the HTTP tunnel mechanism **128** creates a chunked Telnet message at block **166** and forwards the chunked Telnet message through the Internet to the Web server software that in turn passes the chunked Telnet message to the extension **132**. The extension unchunks the chunked Telnet message and forwards to the host system only the Telnet message as originally sent by the emulator to the HTTP tunnel mechanism, at block **168**. As indicated, the sending of Telnet messages and the chunking of the Telnet message into a chunked Telnet message continues without having a new connection established between the Web client and the Web server. Because only one connection is needed during the communication flow, the present invention provides performance comparable to workstations connected through a LAN to a host system (shown in FIG. **1**) and responses that are less variable than response times of prior art methods using translated HTML statements.

One aspect of the present invention is enabling the Web server software **121** to pass the chunked messages through to the extension without waiting for additional chunked messages prior to delivery to the extension **132**. This allows the present invention to tunnel Telnet messages on top of the HTTP protocol using the standard HTTP ports **130**, **130**' and to use the HTTP ports **130**, **130**' as a bi-directional connection for a virtual session between the host system and the Web client.

FIGS. **5**–**7** illustrate formats for the communication flows of FIG. **4** in more detail. First, FIG. **5** illustrates a standard HTTP/1.1 post request format suitable for use in the present invention. The format of the post request is as specified in the HyperText Transfer Protocol HTTP/1.1 specification. The present invention is concerned with the contents of the post request rather than the format of the post request. At line **201**, a request-URI field contains a name for the extension **132** in the Web server. The name in the request-URI field may include a dynamic link library (DLL) or other name of a component depending on the operating system platform. At line **202**, an HTTP-version field must specify version 1.1 or greater. Only version 1.1 or greater allows a transfer-encoding to be chunked as is necessary for the present embodiment of the invention. On line **204**, the transfer-encoding is designated as chunked.

As illustrated in FIG. **4**, at blocks **154**, **158**, **164**, chunking allows the Web client and the host system to exchange a series of messages without having to open a new connection. At line **206** of FIG. **5**, in the present embodiment, the Web server is requested to accept application/octet-stream data. This is not necessary for the purposes of the present invention but is typical for Web servers to expect binary data. At line **208**, in the User Agent field, an identifier for the HTTP tunnel mechanism is provided. The identifier is shown as a non-limiting example and is not necessary to practice the present invention. At line **210**, in the Host field, the name of the Web server is specified. At line **212**, in the Connection field, Keep-Alive is specified so that the connection remains active. However, even with Keep-Alive specified, the connection may be terminated by either the Web client or the Web server after even very brief periods of idleness. In order to achieve a true Keep-Alive connection, the present invention incorporates additional Keep-Alive data in the tunneled data as will be described in greater detail with reference to FIG. **7**.

FIG. **6** illustrates a format for a standard HTTP/1.1 response. Again, the format of the data is as specified in the HTTP/1.1 specification and the present invention is only concerned with the content of the fields. Lines **222**, **224**, and **226** are generated by the Web server in response to the post request (FIG. **5**) that was received and are explained in the HTTP/1.1 specification. Lines **228**, **230**, and **232** are generated by the extension before the response is sent to the Web client. At line **228**, the connection is specified as Keep-Alive as described earlier. Transfer encoding is specified as chunked and the Web server indicates the format of response data as application/octet-stream (i.e., binary data rather than text or other special format data).

FIG. **7** illustrates a format for a chunk suitable for use in the present invention. At line **252**, the first field specifies an ASCII representation of the hex length of the chunk. According to the HTTP 1.1 specification, by allowing each chunk to specify the hex length of the chunk, multiple chunks (messages) may be sent for one HTTP request. However, the last message must provide a unique identifier to indicate when the HTTP request is complete. Typically, the Web server software waits until all the chunks for one message body are received before generating a response. However, because the present invention uses the chunking option in a new manner and does not send a last message, the present invention prevents the Web server software from buffering the chunked messages and instead forces the Web server software to send the chunked messages to the extension as the chunked messages are received. This allows the present invention to provide bi-directional communication without waiting for an entire message and without establishing more than one connection. At line **254**, an end of chunk header indicator is chosen according to the HTTP/1.1 specification. Lines **256** and **258** provide the tunneling aspect of the present invention. First, line **256** provides a data field that may be sent in all chunked packets to keep the connection alive. This Keep-Alive field is necessary because some Web servers will close a connection if additional chunks are not received within a certain time frame. Therefore, in the present invention, the HTTP tunnel mechanism that is running on the Web client generates a chunk with only the Keep-Alive data field and without any additional Telnet data after a predetermined period of inactivity. The sending of only the Keep-Alive data in a chunk allows the connection to remain alive even during periods of inactivity. Line **258** represents the tunneling of the Telnet messages generated by either the host system or the Web client emulator. The Telnet messages generated by either the host system or the Web client emulator are written into the Telnet message field. Line **260** specifies the end of chunk indicator as specified in the HTTP/1.1 specification.

The connection between the Web client and the host system is terminated when the Web client closes the HTTP tunnel mechanism **128**. The extension will also recognize when the host system has prematurely ended the session and will allow the Web server to close the HTTP tunnel.

Although the present embodiment utilizes an emulator that runs as a stand-alone program on the Web client, it will be appreciated that the functionality of the emulator and/or the HTTP tunnel mechanism may be incorporated into a browser. In another embodiment, the Web client may initiate a HEAD response/request before establishing a connection. The initiation of a HEAD response/request is necessary when a Web server needs authentication of the Web client access. The HEAD response provides the Web client with information about authentication that the Web client may incorporate into the post request and the subsequent chunked data.

As one skilled in the art will appreciate, Telnet has many variations such as TN5250, NVT, VT220, and TN3270. These and other proprietary protocols may be used without departing from the scope of this invention. Likewise, the host session may provide a terminal session such as a 5250 type terminal session, a NVT type terminal session, a VT220 type terminal session, and a 3270 type terminal session without departing from the scope of this invention. In addition, although the present embodiment included an IBM host system, it will be appreciated that other host systems, such as Hewlett-Packard and UNIX host systems may be used. Further, even though the embodiment shown provides a virtual session through the Internet, the Internet may be replaced with an intranet, WAN, or other network using HTTP.

While the preferred embodiment of the invention has been illustrated and described, it will be appreciated that various changes can be made therein without departing from the spirit and scope of the invention.

The embodiments of the invention in which an exclusive property or privilege is claimed are defined as follows:

1. A method of providing a persistent HTTP tunnel for a persistent virtual session, the method comprising:

(a) generating a data message complying with a connection-oriented protocol;

(b) embedding the data message within a chunked data message complying with a chunking option of a HTTP specification;

(c) transmitting the chunked data message between a Web client and a Web server via an HTTP connection;

(d) upon receiving any chunked data message at the Web server, parsing the chunked data message and delivering the data message to one endpoint of a connection-oriented session, and upon receiving any chunked data message at the Web client, parsing the chunked data message and delivering the data message to another endpoint of the connection-oriented session; and

(e) interleaving the chunked data messages from the Web client with the chunked data messages from the Web server on the persistent HTTP tunnel.

2. The method of claim **1**, wherein the connection-oriented protocol is a Telnet protocol.

3. The method of claim **1**, wherein the one endpoint of the connection-oriented session is a host system.

4. The method of claim **3**, wherein the other endpoint of the connection-oriented session is a Web client application.

5. The method of claim **4**, wherein the Web client application is a terminal emulator.

6. The method of claim **1**, further comprising (f) transmitting a keep alive chunk message by the Web client to the Web server after a pre-determined time period in which chunked data messages were not transmitted.

7. A method for creating a persistent tunnel between a Web client and a Web server using an HTTP protocol for providing a persistent virtual connection between a host system and the Web client, the method comprising:

establishing a connection between the Web client and the Web server using a chunking option in accordance with an HTTP protocol that allows a series of messages to be sent as chunked messages;

establishing a virtual session between a host system and the Web client through an extension;

transmitting a plurality of host messages from the host system to the Web server, inserting the host message within a chunked host message at the Web server, forwarding the chunked host message to the Web client over the connection, parsing the chunked host message at the Web client and delivering the host message to an application;

transmitting a plurality of Web client messages from the application to a tunneling mechanism on the Web client, inserting the client message within a chunked client message at the Web client, forwarding the chunked client message to the Web server over the connection, parsing the chunked client message at the Web server and delivering the client message to the host system; and

interleaving the chunked data messages from the Web client with the chunked data messages from the Web server on the persistent HTTP tunnel;

wherein the transmitted chunked client message is forwarded to the extension before the Web server is forwarded the entire plurality of chunked client messages.

8. The method of claim **7**, wherein the application is a terminal emulator.

* * * * *

US 20030009571A1

(19) **United States**

(12) **Patent Application Publication** (10) **Pub. No.: US 2003/0009571 A1**

Bavadekar (43) **Pub. Date: Jan. 9, 2003**

(54) **SYSTEM AND METHOD FOR PROVIDING TUNNEL CONNECTIONS BETWEEN ENTITIES IN A MESSAGING SYSTEM**

(76) Inventor: **Shailesh S. Bavadekar**, Sunnyvale, CA (US)

Correspondence Address:
Robert C. Kowert
Conley, Rose, & Tayon, P.C.
P.O. Box 398
Austin, TX 78767-1400 (US)

(21) Appl. No.: **09/894,318**

(22) Filed: **Jun. 28, 2001**

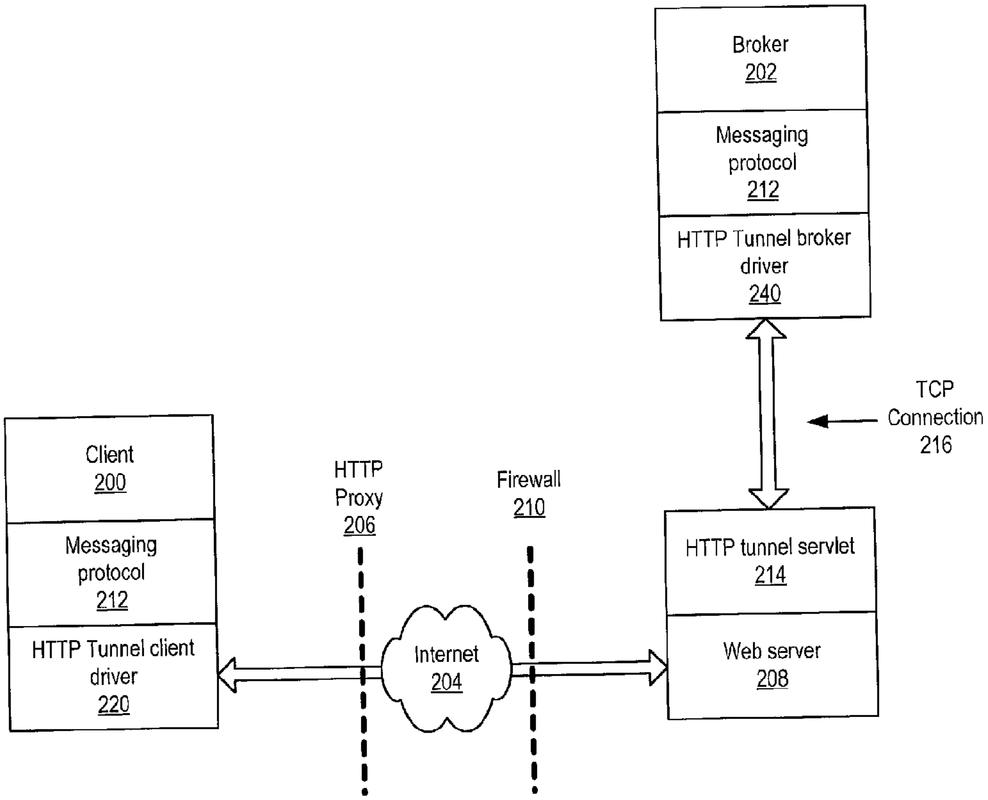
Publication Classification

(51) **Int. Cl.**^{7} **G06F 15/16**

(52) **U.S. Cl.** **709/230**; 709/203

(57) **ABSTRACT**

A system and method for providing HTTP tunnel connections between entities such as clients and servers in a messaging system is described. An HTTP tunnel connection layer is described that may be used to provide reliable, full duplex virtual connections between entities (e.g. clients and brokers) in a distributed application environment using a messaging system. Also described is a novel HTTP tunneling protocol that may be used by the HTTP tunnel connection layer. The HTTP tunnel connection layer may be used by clients to access messaging servers through proxy servers and firewalls, thus expanding the scope of from where clients can access brokers. Using this layer, brokers as well as clients may initiate messaging system messages. This layer may also provide guaranteed data delivery with correct sequencing even in case of a failure on the network. This layer may also provide end-to-end flow control.



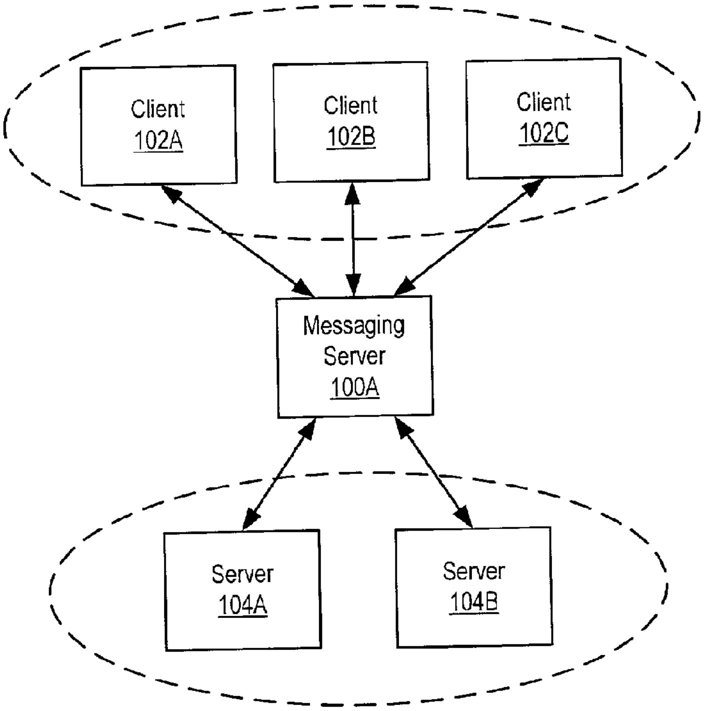


Figure 1 - Prior Art

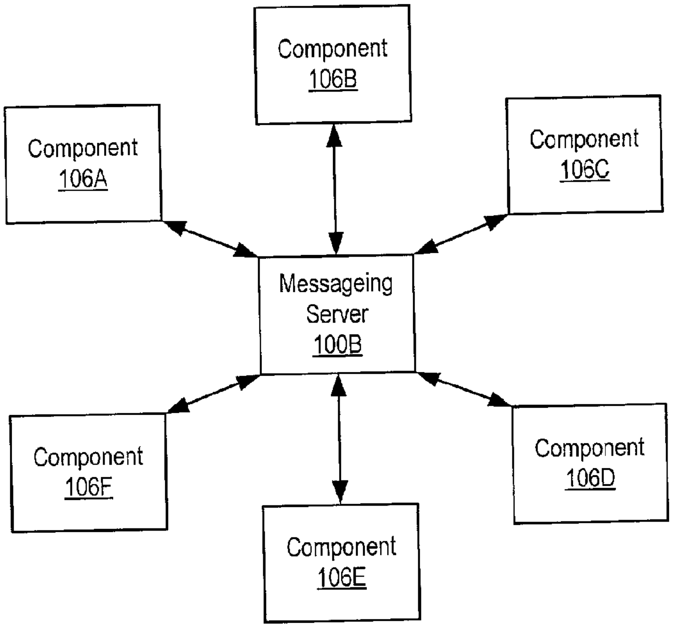


Figure 2 - Prior Art

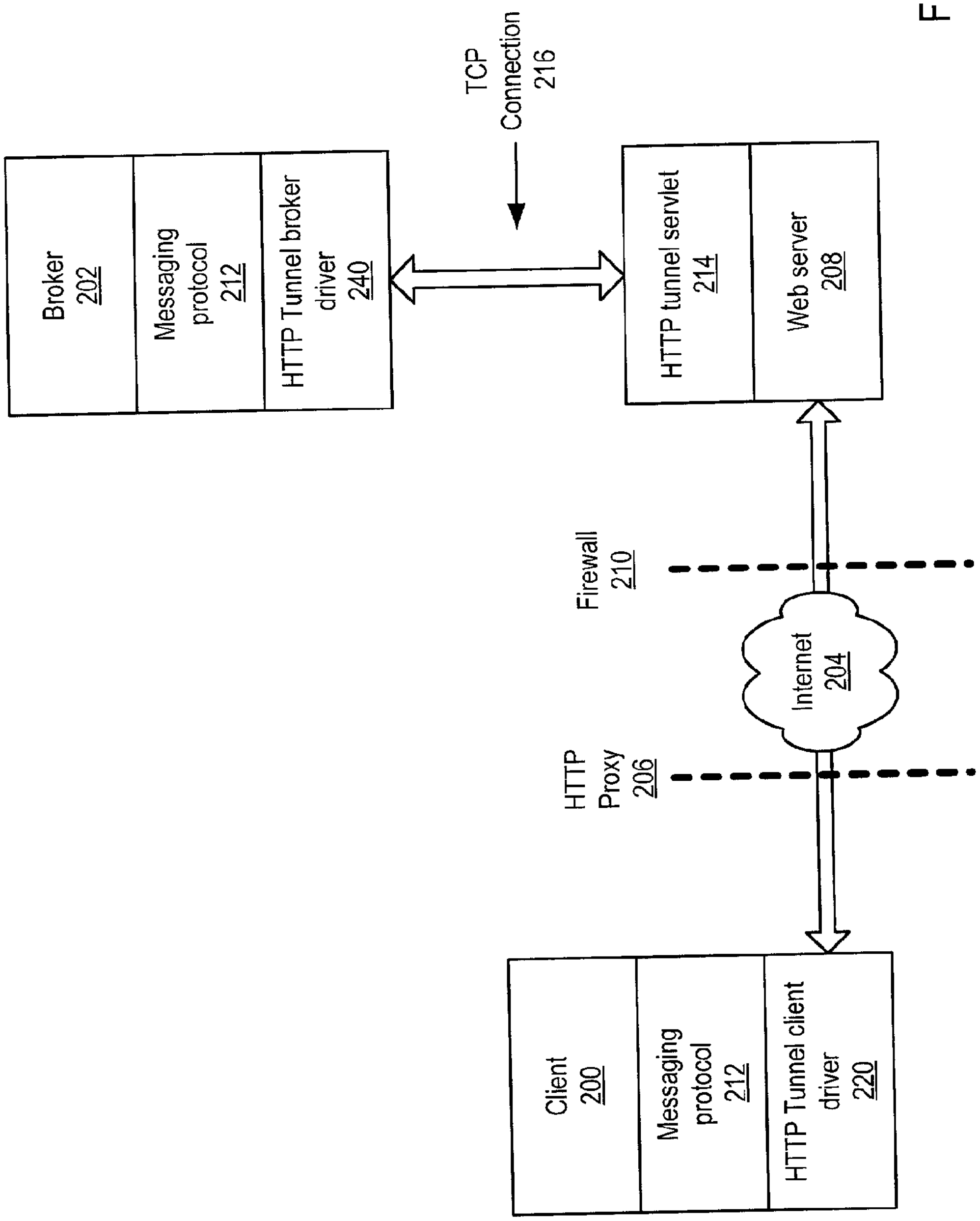

Broker
202
Messaging protocol
212
HTTP Tunnel broker driver
240
TCP Connection
216
HTTP tunnel servlet
214
Web server
208
Firewall
210
Internet
204
HTTP Proxy
206
Client
200
Messaging protocol
212
HTTP Tunnel client driver
220

Figure 3A

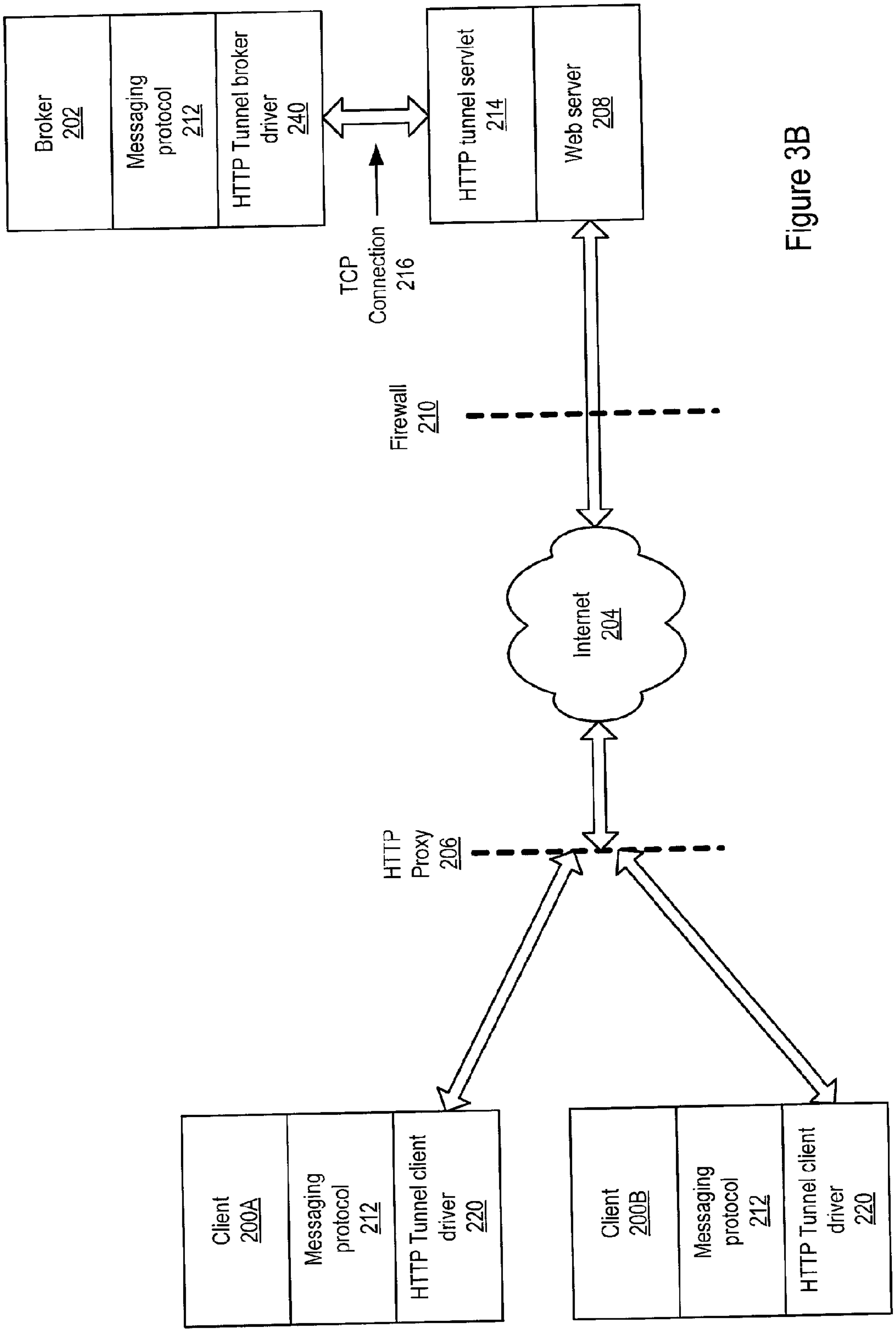

Broker
202
Messaging protocol
212
HTTP Tunnel broker driver
240
HTTP tunnel servlet
214
Web server
208
TCP Connection
216
Firewall
210
Internet
204
HTTP Proxy
206
Client
200A
Messaging protocol
212
HTTP Tunnel client driver
220
Client
200B
Messaging protocol
212
HTTP Tunnel client driver
220

Figure 3B

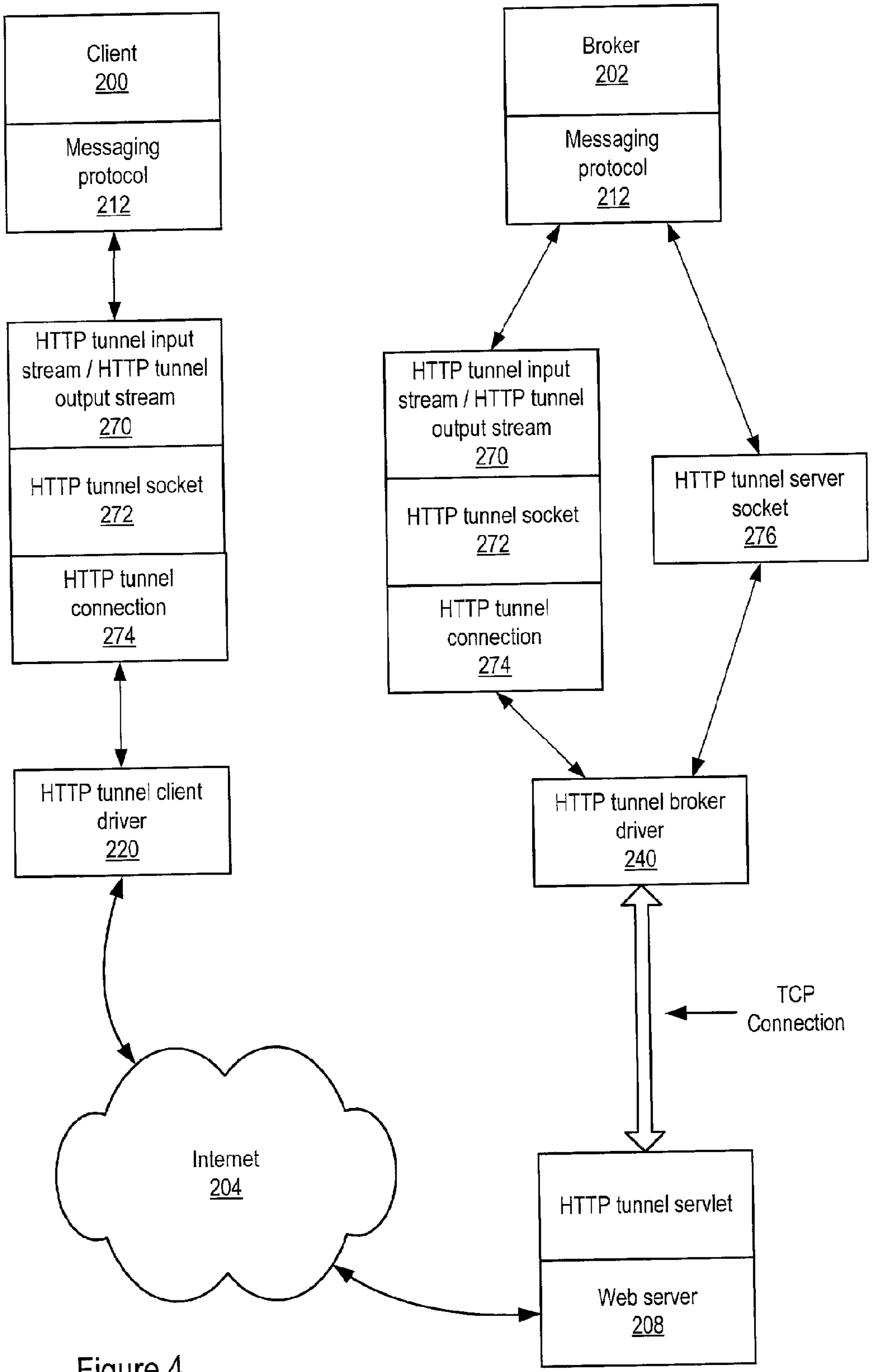

Client
200
Messaging protocol
212
HTTP tunnel input stream / HTTP tunnel output stream
270
HTTP tunnel socket
272
HTTP tunnel connection
274
HTTP tunnel client driver
220
Internet
204
Broker
202
Messaging protocol
212
HTTP tunnel input stream / HTTP tunnel output stream
270
HTTP tunnel socket
272
HTTP tunnel connection
274
HTTP tunnel server socket
276
HTTP tunnel broker driver
240
TCP Connection
HTTP tunnel servlet
Web server
208

Figure 4

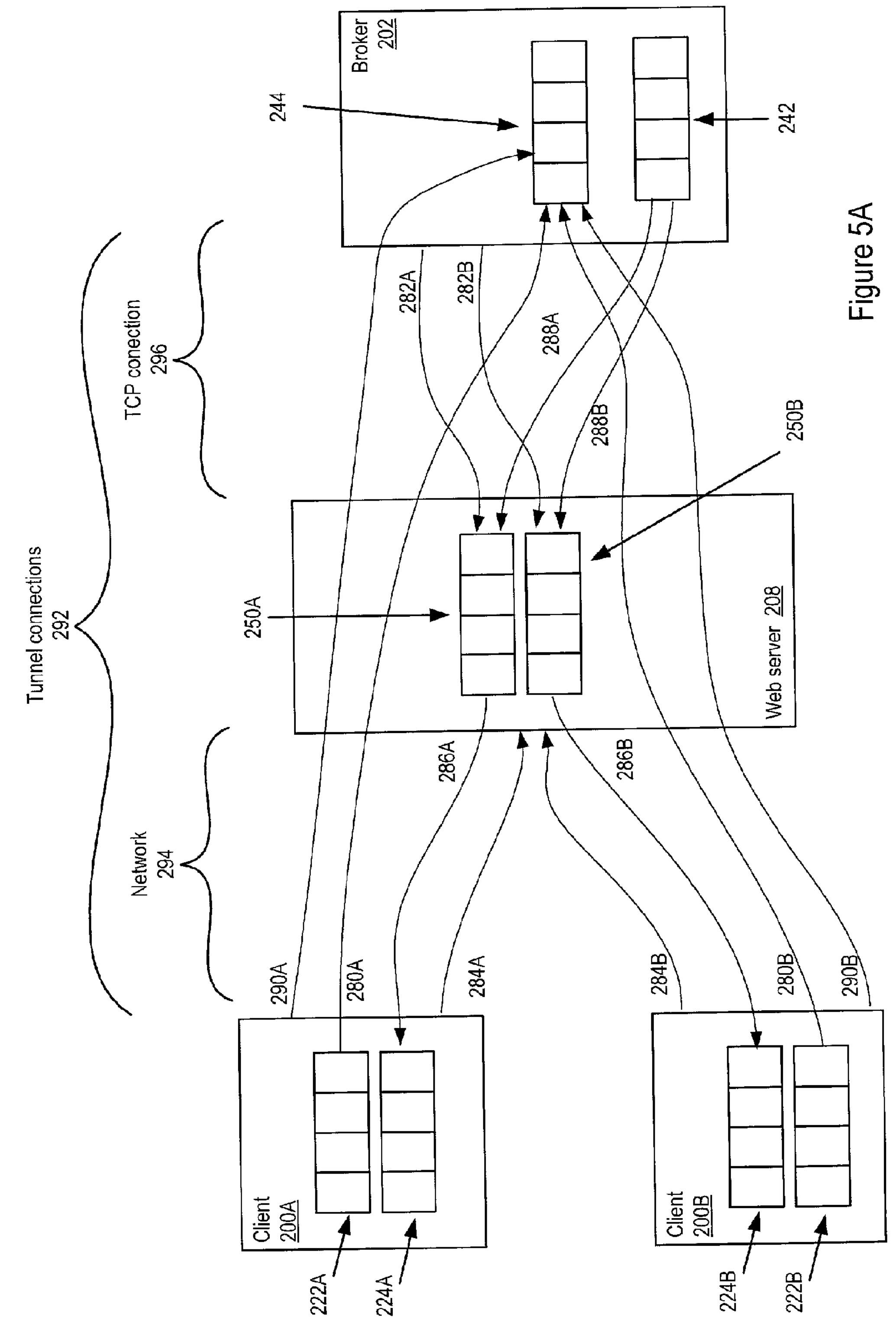

Broker 202
244
242
Tunnel connections 292
TCP conection 296
Network 294
282A
282B
288A
288B
250A
250B
Web server 208
286A
286B
290A
280A
284A
284B
280B
290B
Client 200A
Client 200B
222A
224A
224B
222B

Figure 5A

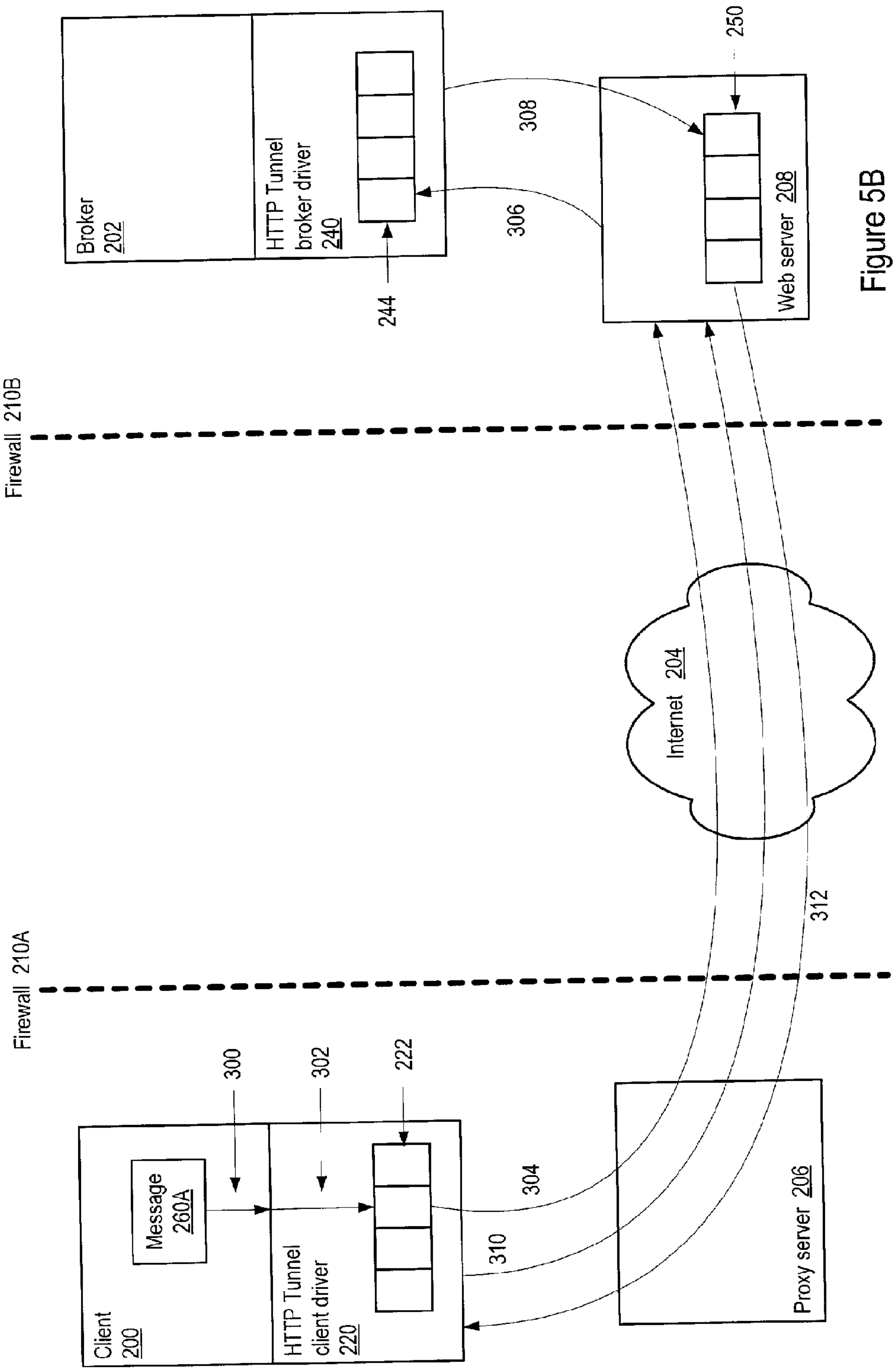

Broker
202
HTTP Tunnel
broker driver
240
244
250
308
306
Web server 208
Firewall 210B
Internet 204
312
Firewall 210A
300
302
222
Client
200
Message
260A
HTTP Tunnel
client driver
220
304
310
Proxy server 206

Figure 5B

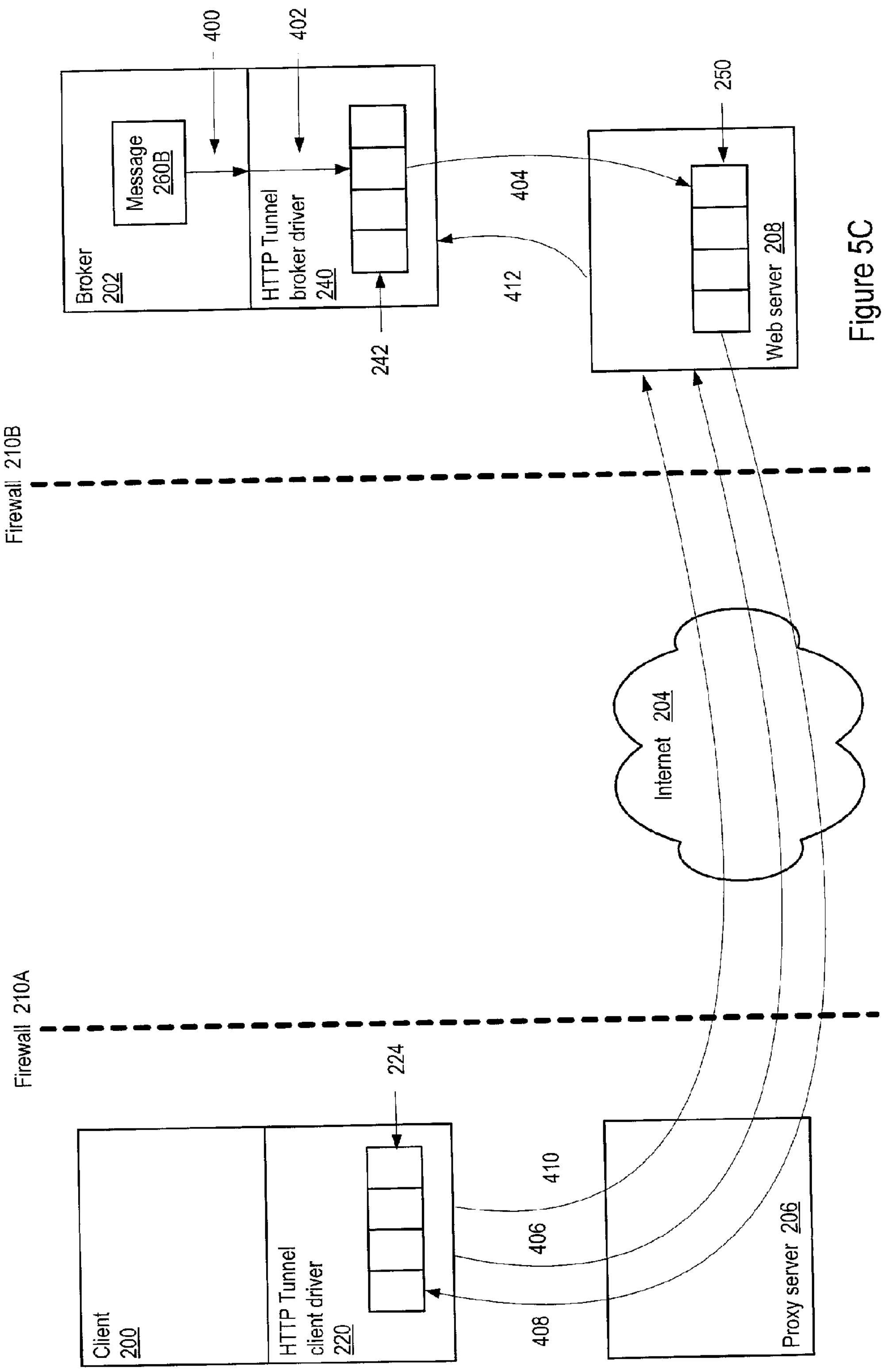

Firewall 210B
Firewall 210A
Broker 202
Message 260B
HTTP Tunnel broker driver 240
400
402
242
404
412
Web server 208
250
Internet 204
Client 200
HTTP Tunnel client driver 220
224
410
406
408
Proxy server 206

Figure 5C

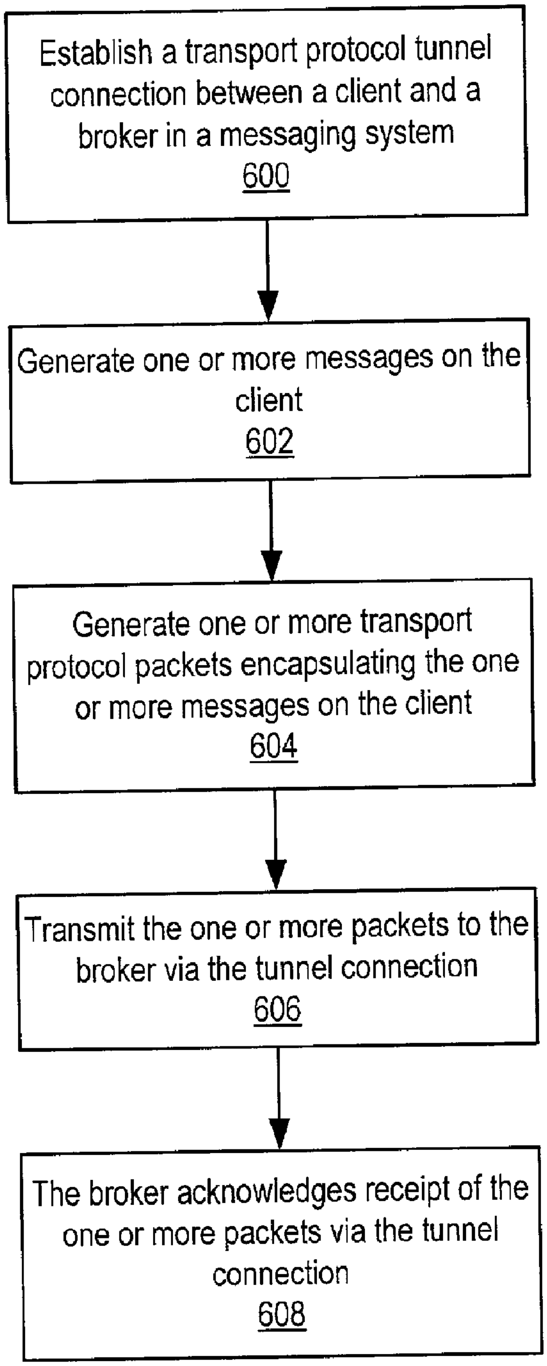

Establish a transport protocol tunnel connection between a client and a broker in a messaging system
600
Generate one or more messages on the client
602
Generate one or more transport protocol packets encapsulating the one or more messages on the client
604
Transmit the one or more packets to the broker via the tunnel connection
606
The broker acknowledges receipt of the one or more packets via the tunnel connection
608

Figure 6A

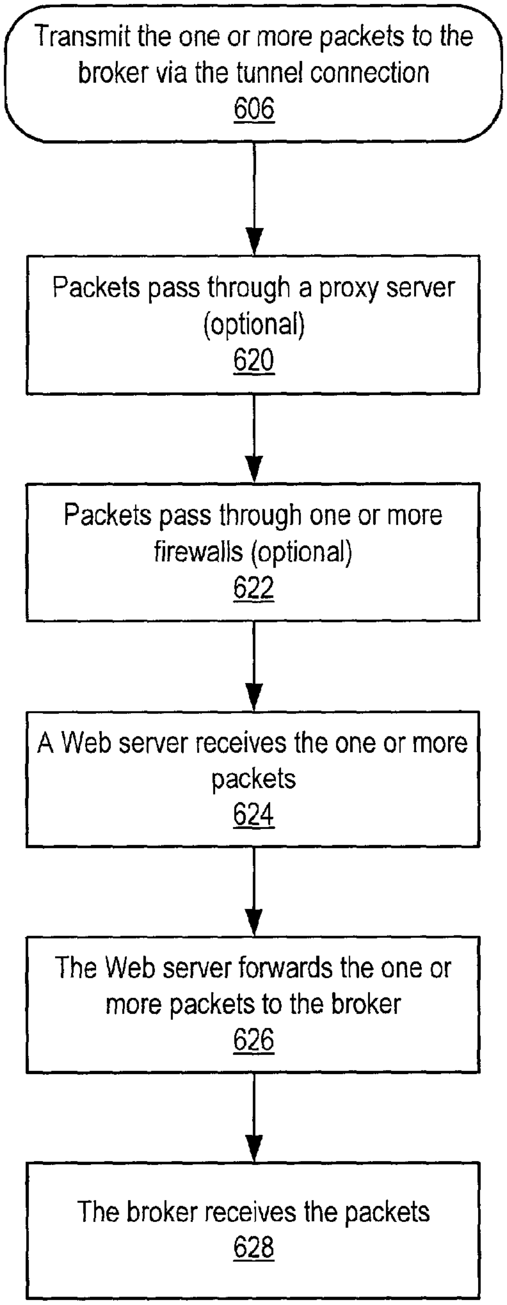

Transmit the one or more packets to the broker via the tunnel connection
606
Packets pass through a proxy server (optional)
620
Packets pass through one or more firewalls (optional)
622
A Web server receives the one or more packets
624
The Web server forwards the one or more packets to the broker
626
The broker receives the packets
628

Figure 6B

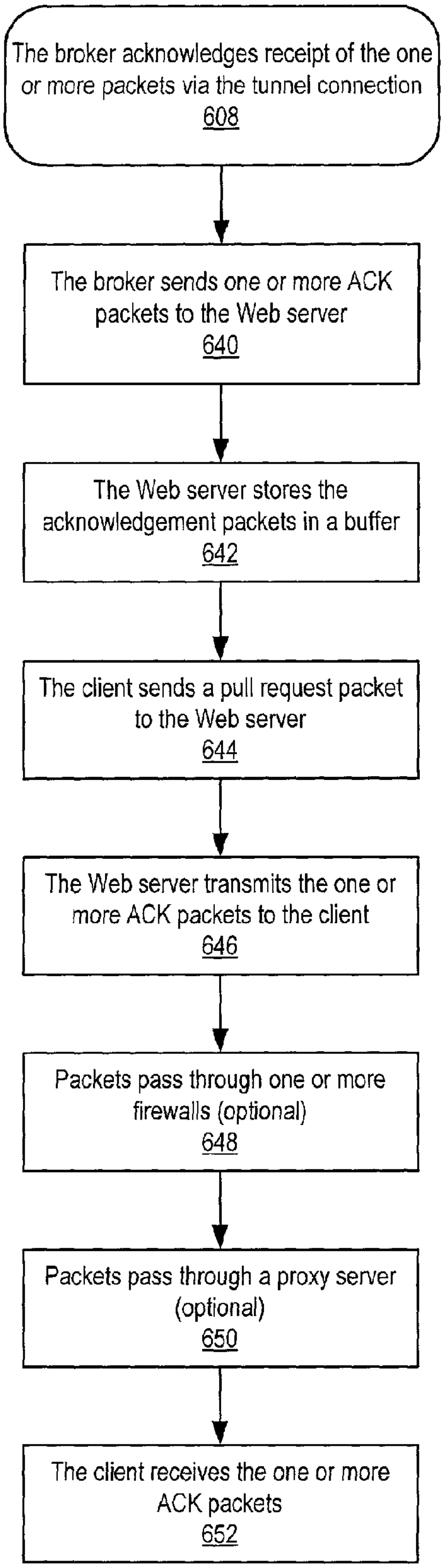

The broker acknowledges receipt of the one or more packets via the tunnel connection
608
The broker sends one or more ACK packets to the Web server
640
The Web server stores the acknowledgement packets in a buffer
642
The client sends a pull request packet to the Web server
644
The Web server transmits the one or more ACK packets to the client
646
Packets pass through one or more firewalls (optional)
648
Packets pass through a proxy server (optional)
650
The client receives the one or more ACK packets
652

Figure 6C

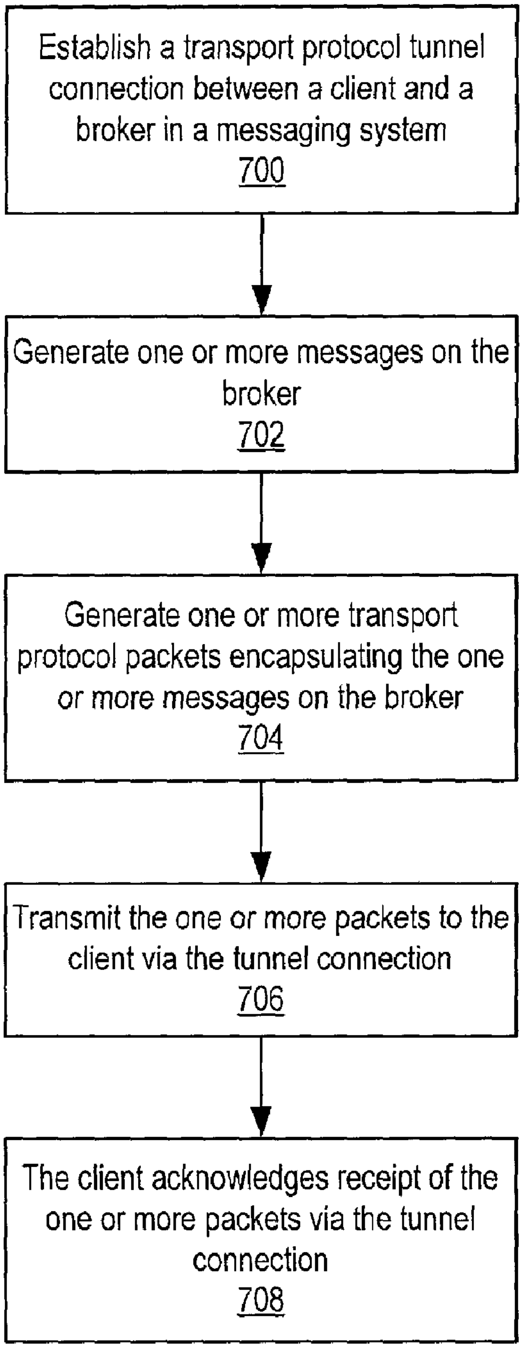

Establish a transport protocol tunnel connection between a client and a broker in a messaging system
700
Generate one or more messages on the broker
702
Generate one or more transport protocol packets encapsulating the one or more messages on the broker
704
Transmit the one or more packets to the client via the tunnel connection
706
The client acknowledges receipt of the one or more packets via the tunnel connection
708

Figure 7A

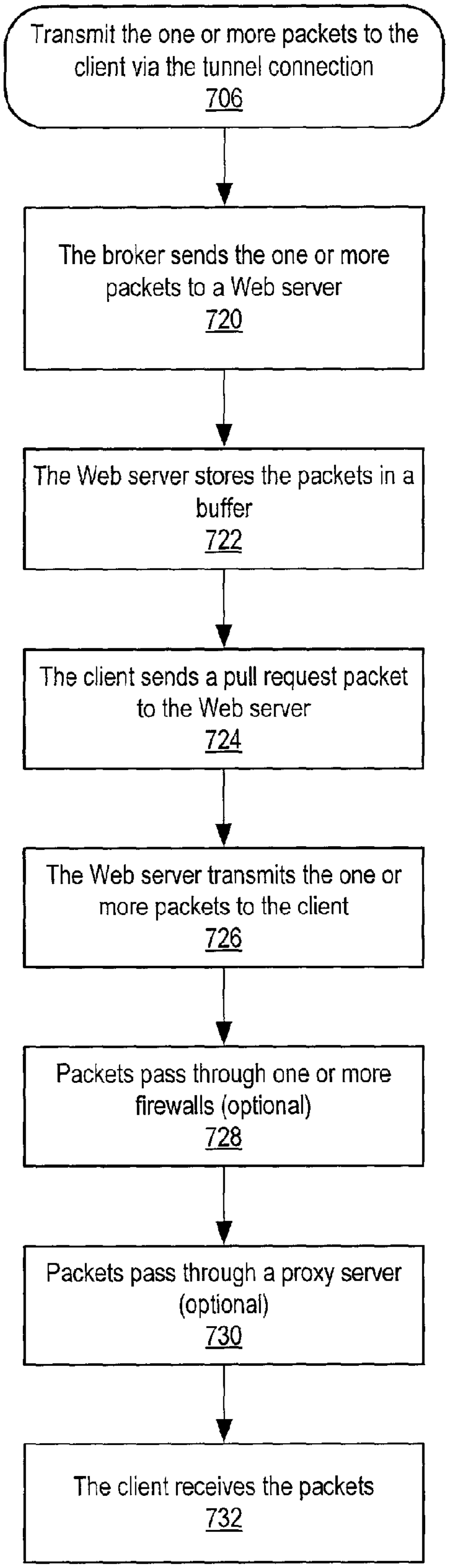

Transmit the one or more packets to the client via the tunnel connection
706
The broker sends the one or more packets to a Web server
720
The Web server stores the packets in a buffer
722
The client sends a pull request packet to the Web server
724
The Web server transmits the one or more packets to the client
726
Packets pass through one or more firewalls (optional)
728
Packets pass through a proxy server (optional)
730
The client receives the packets
732

Figure 7B

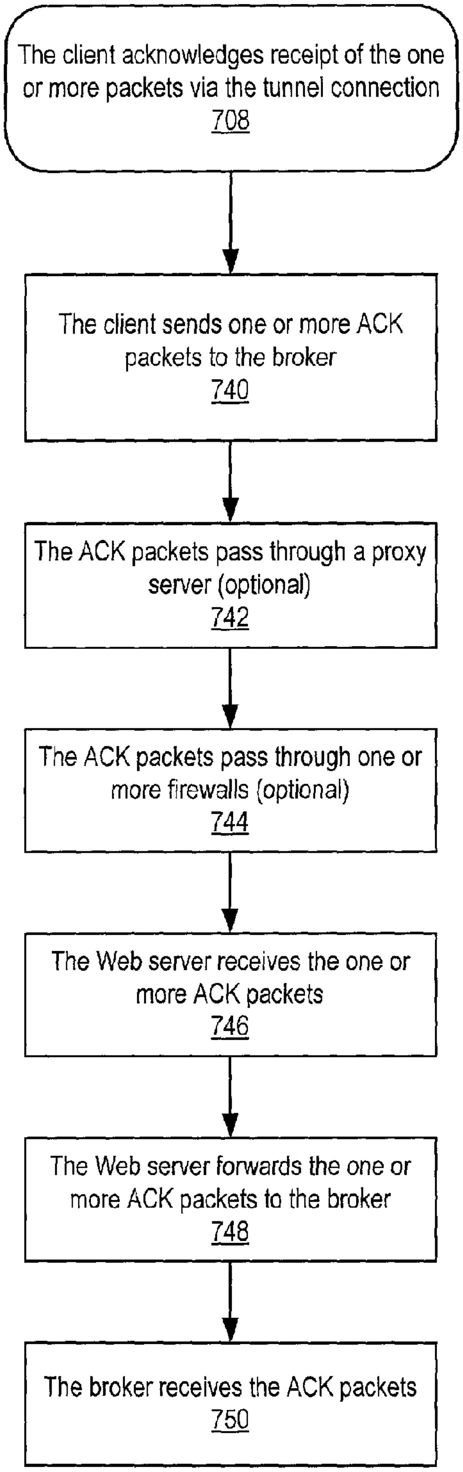

The client acknowledges receipt of the one or more packets via the tunnel connection
708
The client sends one or more ACK packets to the broker
740
The ACK packets pass through a proxy server (optional)
742
The ACK packets pass through one or more firewalls (optional)
744
The Web server receives the one or more ACK packets
746
The Web server forwards the one or more ACK packets to the broker
748
The broker receives the ACK packets
750

Figure 7C

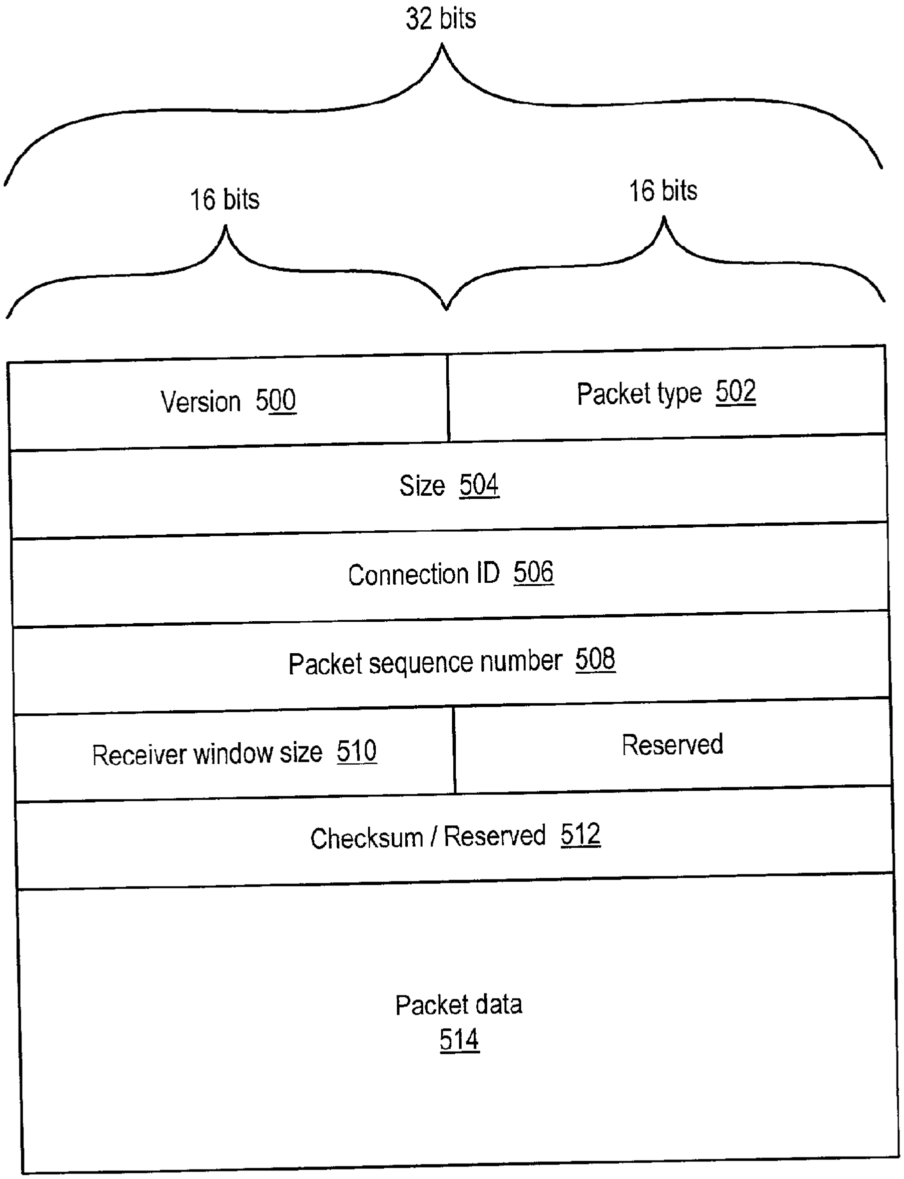


Figure 8

Exemplary tunneling packet format

SYSTEM AND METHOD FOR PROVIDING TUNNEL CONNECTIONS BETWEEN ENTITIES IN A MESSAGING SYSTEM

BACKGROUND OF THE INVENTION

[0001] 1. Field of the Invention

[0002] This invention relates to computers and networks of computers, and more particularly to a system and method for providing transport protocol tunnel connections between entities or nodes such as clients and servers in a messaging system.

[0003] 2. Description of the Related Art

[0004] Messaging is playing an increasingly important role in computing. Its advantages are a natural result of several factors: the trend toward peer-to-peer computing, greater platform heterogeneity, and greater modularity, coupled with the trend away from synchronous communication between processes. The common building block of a messaging service is the message. Messages are specially formatted data describing events, requests, and replies that are created by and delivered to computer programs. Messages contain formatted data with specific meanings. Messaging is the exchange of messages to a messaging server, which acts as a message exchange program for client programs. A messaging server is a middleware program that handles messages that are sent by client programs for use by other programs. Typically, client programs access the functionality of the messaging system using a messaging application program interface (application program interface). A messaging server can usually queue and prioritize messages as needed, and thus saves each of the client programs from having to perform these services. Rather than communicate directly with each other, the components in an application based around a message service send messages to a message server. The message server, in turn, delivers the messages to the specified recipients

[0005] There are two major messaging system models: the point-to-point model and the publish and subscribe model. Messaging allows programs to share common message-handling code, to isolate resources and interdependencies, and to easily handle an increase in message volume. Messaging also makes it easier for programs to communicate across different programming environments (languages, compilers, and operating systems) since the only thing that each environment needs to understand is the common messaging format and protocol. The messages involved exchange crucial data between computers—rather than between users—and contain information such as event notification and service requests. IBM's MQSeries and iPlanet Message Queue are examples of products that provide messaging interfaces and services.

[0006] **FIG. 1** illustrates a typical messaging-based application. This application is a modification of the traditional client/server architecture. The major difference is the presence of a messaging server **100**A between client **102** and server **104** layers. Thus, rather than communicating directly, clients **102** and servers **104** communicate via the messaging server **100**A. The addition of the messaging server **100**A adds another layer to the application, but it greatly simplifies the design of both the clients **102** and the servers **104** (they are no longer responsible for handling communications issues), and it also enhances scalability. Note that servers in a messaging system may also be referred to as "brokers".

[0007] **FIG. 2** illustrates another messaging-based application based on point-to-point architecture. This type of application almost demands a centralized messaging server **100**B. Without one, each component **106** would be responsible for creating and maintaining connections with the other components **106**. A possible alternative approach would be to architect the system around a communication bus, but this would still leave each component **106** in charge of message delivery issues.

[0008] Java Message Service (JMS)

[0009] Java Message Service (JMS) is an application program interface (API) from Sun Microsystems that supports messaging between computers in a network. JMS provides a common interface to standard messaging protocols and also to special messaging services in support of Java programs. Sun advocates the use of the JMS for anyone developing Java applications, which can be run from any major operating system platform. Using the JMS interface, a programmer can invoke the messaging services of IBM's MQSeries, Progress Software's SonicMQ, and other messaging product vendors.

[0010] The JMS API may:

[0011] Provide a single, unified message API

[0012] Provide an API suitable for the creation of messages that match the format used by existing, non-JMS applications

[0013] Support the development of heterogeneous applications that span operating systems, platforms, architectures, and computer languages

[0014] Support messages that contain serialized Java objects

[0015] Support messages that contain eXtensible Markup Language (XML) pages

[0016] Allow messages to be prioritized

[0017] Deliver messages either synchronously or asynchronously

[0018] Guarantee messages are delivered once and only once

[0019] Support message delivery notification

[0020] Support message time-to-live

[0021] Support transactions

[0022] The JMS API is divided into two nearly identical pieces. One implements a point-to-point model of messaging, and the other implements a publish and subscribe model of messaging. Each of these models is called a domain. The APIs are almost identical between the domains. The separation of the API into two domains relieves vendors that support only one messaging model from providing facilities their product doesn't natively support.

[0023] Enterprise Messaging Systems

[0024] Enterprise messaging systems may be developed using a messaging service such as JMS. An enterprise messaging system may be used to integrate distributed, loosely coupled applications/systems in a way that provides for dynamic topologies of cooperating systems/services. Enterprise messaging systems typically need to address common messaging related problems such as:

[0025] Guaranteed message delivery (e.g. persistence, durable interests, "at least once" and "once and only once" message delivery guarantees, transactions etc). Messages from one component to another must not be lost due to network or system failure. This means the system must be able to guarantee that a message is successfully delivered.

[0026] Asynchronous delivery. For large numbers of components to be able to exchange messages simultaneously, and support high density throughputs, the sending of a message cannot depend upon the readiness of the consumer to immediately receive it. If a consumer is busy or offline, the system must allow for a message to be sent and subsequently received when the consumer is ready. This is known as asynchronous message delivery, popularly known as store-and-forward messaging.

[0027] Various message delivery models (e.g. publish and subscribe or point-to-point).

[0028] Transport independence.

[0029] Leveraging an enterprise messaging system in developing business solutions allows developers to focus on their application/business logic rather than on implementing the underlying messaging layer.

[0030] iPlanet E-Commerce Solutions' iMQ (iplanet Message Queue), formerly offered by Sun Microsystems as JMQ (Java Message Queue) is an example of an enterprise messaging system, and was developed to be JMS-compliant. iMQ may use a "hub and spoke" architecture. Clients use an iMQ client library to exchange messages with an iMQ message server (also referred to as a "broker").

[0031] In an enterprise messaging system, clients exchange messages with a messaging server using a message exchange protocol. The messaging server then may route the messages based upon properties of the messages. Typically, the message exchange protocol requires a direct, fully bi-directional reliable transport connection between the client and the messaging server, such as a TCP (Transport Control Protocol) or SSL (Secure Sockets Layer) connection, which can be used only if the client and the messaging server both reside on the "intranet" (i.e. on the same side of a firewall).

[0032] Hypertext Transfer Protocol

[0033] The Hypertext Transfer Protocol (HTTP) is a set of rules for exchanging files (text, graphic images, sound, video, and other multimedia files) on the World Wide Web. HTTP may also be used on an intranet. Relative to the TCP/IP suite of protocols (which are the basis for information exchange on the Internet), HTTP is an application protocol that is implemented over TCP/IP. HTTP was designed as a stateless request-response mechanism.

[0034] A Web server is a program that, using the client/server model and HTTP, serves the files that form Web pages to Web users (whose computers contain HTTP clients that forward their requests). Every computer on the Internet that contains a Web site must have a Web server. A Web server machine may include, in addition to the Hypertext Markup Language (HTML) and other files it can serve, an HTTP daemon, a program that is designed to wait for HTTP requests and handle them when they arrive. A Web browser is an example of an HTTP client that sends requests to server machines. When a browser user enters file requests by either "opening" a Web file by typing in a URL (Uniform Resource Locator) or clicking on a hypertext link, the browser builds an HTTP request and sends it to the Internet Protocol (IP) address indicated by the URL. The HTTP daemon in the destination server machine receives the request and, after any necessary processing, the requested file is returned.

[0035] Tunneling

[0036] Tunneling may be defined as the encapsulation of a protocol A within protocol B, such that A treats B as though it were a data link layer. Tunneling may be used to get data between administrative domains that use a protocol that is not supported by the Internet connecting those domains. A "tunnel" is a particular path that a given message or file might travel through the Internet.

[0037] Proxy Servers and Firewalls.

[0038] In an enterprise that uses the Internet, a proxy server is a server that acts as an intermediary between a workstation user and the Internet so that the enterprise can ensure security, administrative control, and caching service. A proxy server may be associated with or part of a gateway server that separates the enterprise network from the outside network and a firewall server that protects the enterprise network from outside intrusion.

[0039] A firewall is a set of related programs, usually located at a network gateway server, that protects the resources of a private network from users from other networks. An enterprise with an intranet that allows its workers access to the wider Internet installs a firewall to prevent outsiders from accessing its own private data resources and for controlling what outside resources its own users have access to.

SUMMARY OF THE INVENTION

[0040] A system and method for providing transport protocol tunnel connections between entities such as clients and servers in a messaging system is described. A transport protocol tunnel connection layer is described that may be used to provide reliable, full duplex virtual connections between entities (e.g. clients and servers) in a distributed application environment using a messaging system. This layer may be used by clients to access messaging servers (referred to as brokers) through proxy servers and firewalls, thus expanding the scope of from where clients can access brokers. This layer allows for information flow in both directions between the client and the server. Using this layer, brokers as well as clients may initiate messages in the distributed application environment. This layer may also provide guaranteed data delivery with correct sequencing even in case of a failure on the network. This layer may also provide end-to-end flow control.

[0041] The transport protocol tunnel connection layer may provide a connection-oriented, bi-directional byte stream service between nodes in a messaging system. Application data, including messages, may be carried as transport protocol packet payloads. The transport protocol tunnel connection layer allows messaging system clients to access messaging system brokers using the transport protocol instead of using direct TCP connections. This enables the clients to access the brokers through firewalls. Also, with the help of a transport protocol proxy, the clients may access the messaging service even when there is no direct IP connectivity with the broker.

[0042] Using embodiments of the transport protocol tunnel connection layer, a transport protocol tunnel connection between the client and the broker in the messaging system may be established. The client may then generate one or more messaging system messages. In one embodiment, the generated messages may then be stored in a client-side transmit buffer. One or more transport protocol packets encapsulating the one or more messages may then be generated on the client. In one embodiment, a client-side tunnel connection driver may generate the packets and include the messages as payloads of the packets. The one or more transport protocol packets may then be transmitted to the broker via the tunnel connection. In one embodiment, the client-side tunnel connection driver may handle the transmission of the packets.

[0043] A Web server may then receive the one or more transport protocol packets. The Web server may then forward the received transport protocol packets to the broker. In one embodiment, the packets may be forwarded to the broker via a TCP connection serving as one segment of the transport protocol tunnel connection between the Web server and the broker. In one embodiment, a transport protocol tunnel servlet on the Web server node may serve as an interface between the Web server and the TCP connection.

[0044] The broker may receive the transport protocol packets from the Web server. In one embodiment, a broker-side transport protocol tunnel driver may receive the packets. In one embodiment, the broker may extract the messaging system messages from the transport protocol packets and store the messages in a broker-side receive buffer. In another embodiment, the entire transport protocol packet may be stored in the broker-side receive buffer.

[0045] The broker may acknowledge receipt of the packets by sending one or more acknowledgement (ACK) packets to the client via the tunnel connection. The broker may generate and send one or more acknowledgement (ACK) packets to the Web server. In one embodiment, the broker may store ACK packets in a broker-side transmit buffer. In one embodiment, the ACK packets may be sent to the Web server over the TCP connection. In one embodiment, the broker-side transport protocol tunnel driver may handle the transmission of the ACK packets to the Web server.

[0046] The Web server may then receive the ACK packets. In one embodiment, a transport protocol tunnel servlet may receive the packets for the Web server. The Web server may store the acknowledgement packets in a transport protocol packet buffer. At some point, the client may send a transport protocol pull request packet to the Web server. In one embodiment, the client may periodically send pull requests to the Web server. In one embodiment, a separate thread on the client may handle periodically sending pull requests. The Web server may transmit to the client the one or more transport protocol ACK packets stored in the transport protocol packet buffer associated with the client in response to receiving the pull request packet. The client may then receive the one or more ACK packets. The ACK packets may serve to acknowledge the receipt of the transmitted data so that the sender may free its transmit buffers.

[0047] Using embodiments of the transport protocol tunnel connection layer, messaging system messages may be generated on a broker and sent to a client. In one embodiment, the generated messages may be stored in a broker-side transmit buffer. One or more transport protocol packets encapsulating the one or more messages may then be generated on the broker. In one embodiment, a broker-side transport protocol tunnel connection driver may generate the transport protocol packets and include the messages as payloads of the packets. The one or more transport protocol packets may then be transmitted to the client via the transport protocol tunnel connection. In one embodiment, the broker-side transport protocol tunnel connection driver may handle the transmission of the packets. In one embodiment, the transport protocol packets may be sent to a Web server. In one embodiment, the transport protocol packets may be sent to the Web server over a TCP connection.

[0048] The Web server may receive the transport protocol packets and store the received packets in a transport protocol packet buffer. In one embodiment, a transport protocol tunnel servlet may receive the packets for the Web server. At some point, the client may send a transport protocol pull request packet to the Web server. The Web server may transmit to the client one or more transport protocol packets stored in the transport protocol packet buffer associated with the client in response to receiving the pull request packet.

[0049] The client may then receive the one or more transport protocol packets. In one embodiment, the client may store the received transport protocol packets in a client-side receive buffer. After receiving the transport protocol packets, the client may acknowledge receipt of the packets by sending one or more acknowledgement (ACK) packets to the broker via the transport protocol tunnel connection. The Web server may receive the ACK packets and forward the received ACK packets to the broker. In one embodiment, the ACK packets may be transmitted to the broker via a TCP connection between the Web server and the broker that serves as one segment of the HTTP tunnel connection. The broker may receive the ACK packets from the Web server. In one embodiment, the received ACK packets may be stored in a broker-side receive buffer.

[0050] In one embodiment, one Web server and one tunnel servlet may be used by two or more clients to communicate to a broker via tunnel connections. In this embodiment, the HTTP tunnel servlet may multiplex transport protocol packets from the two or more clients onto the TCP connection. In one embodiment, the tunnel servlet may extract the messaging system message information from received transport protocol packets and send only the message information to the broker over the TCP connection. In one embodiment, there may be one broker-side receive buffer for each tunnel connection. In another embodiment, a single receive buffer may be used for two or more tunnel connections.

[0051] Transport protocol packets may optionally pass through a proxy server and one or more firewalls. For example, transport protocol packet flow from a client to a broker may pass through a proxy server, through a firewall onto the Internet, through another firewall to a Web server, and from the Web server over a TCP connection to the broker. Packets flowing in the opposite direction (from the broker to the client) take the reverse path.

[0052] In one embodiment, transport protocol packets transmitted on the tunnel connection may include message sequence information configured for use by the receiver in processing received messages in the correct sequence. In one embodiment, flow control may be applied to the sending of messages from a sender to a receiver. In one embodiment, the receiver may inform the sender of available space in a receive buffer to store incoming messages. In one embodiment, upon establishment of a tunnel connection, each side (both clients and brokers can be senders and/or receivers) may inform the other of its receive buffer size. In one embodiment, ACK packets sent to a sender by a receiver may each include the currently available space in the receive buffer. Thus, the sender can keep track of the current receive capacity of the receiver.

BRIEF DESCRIPTION OF THE DRAWINGS

[0053] **FIG. 1** illustrates a prior art messaging-based application based upon client/server architecture;

[0054] **FIG. 2** illustrates a prior art messaging-based application based on point-to-point architecture;

[0055] **FIG. 3A** illustrates a client-server messaging system implementing a transport protocol tunnel connection layer according to one embodiment;

[0056] **FIG. 3B** illustrates a client-server messaging system implementing a transport protocol tunnel connection layer with multiple clients accessing a broker through a Web server according to one embodiment;

[0057] **FIG. 4** illustrates the architecture of a client-server messaging system implementing a transport protocol tunnel connection layer according to one embodiment;

[0058] **FIG. 5A** illustrates the routing of transport protocol packets between clients and a broker on transport protocol tunnel connections according to one embodiment;

[0059] **FIG. 5B** illustrates the process of sending a message from a client to a broker via a transport protocol tunnel connection according to one embodiment;

[0060] **FIG. 5C** illustrates the process of sending a message from a broker to a client via a transport protocol tunnel connection according to one embodiment;

[0061] **FIGS. 6A** is a flowchart of a method for sending messages from a messaging system client to a messaging system broker over a transport protocol tunnel connection layer according to one embodiment; **FIG. 6B** is a flowchart of a method for transmitting one or more packets from a client to a broker via a transport protocol tunnel connection according to one embodiment;

[0062] **FIG. 6C** is a flowchart of a method for a broker to acknowledge to a client the receipt of transport protocol packets via the transport protocol tunnel connection according to one embodiment;

[0063] **FIG. 7A** is a flowchart of a method for sending messages from a messaging system client to a messaging system broker over a transport protocol tunnel connection layer according to one embodiment;

[0064] **FIG. 7B** is a flowchart of a method for transmitting one or more packets from a broker to a client via a transport protocol tunnel connection according to one embodiment;

[0065] **FIG. 7C** is a flowchart of a method for a client to acknowledge to a broker the receipt of transport protocol packets via a transport protocol tunnel connection according to one embodiment; and

[0066] **FIG. 8** illustrates an exemplary transport protocol packet format that may be used in the transport protocol tunnel connection layer according to one embodiment.

[0067] While the invention is described herein by way of example for several embodiments and illustrative drawings, those skilled in the art will recognize that the invention is not limited to the embodiments or drawings described. It should be understood, that the drawings and detailed description thereto are not intended to limit the invention to the particular form disclosed, but on the contrary, the intention is to cover all modifications, equivalents and alternatives falling within the spirit and scope of the present invention as defined by the appended claims. The headings used herein are for organizational purposes only and are not meant to be used to limit the scope of the description or the claims. As used throughout this application, the word "may" is used in a permissive sense (i.e., meaning having the potential to), rather than the mandatory sense (i.e., meaning must). Similarly, the words "include", "including", and "includes" mean including, but not limited to.

DETAILED DESCRIPTION OF EMBODIMENTS OF THE INVENTION

[0068] A system and method for providing transport protocol tunnel connections between entities such as clients and servers in a messaging system is described. A transport protocol tunnel connection layer is described that may be used to provide reliable, full duplex virtual connections between entities (e.g. clients and servers) in a distributed application environment using a messaging system. This layer may be used by clients to access messaging servers (referred to as brokers) through proxy servers and firewalls, thus expanding the scope of from where clients can access brokers. Using this layer, brokers as well as clients may initiate messages in the distributed application environment. This layer may also provide guaranteed data delivery with correct sequencing even in case of a failure on the network. This layer may also provide end-to-end flow control. The transport protocol tunnel connection layer may be used to simulate virtual connections that have a similar contract with the upper layers (e.g. clients and brokers) as a TCP connection. Thus, applications can be developed in terms of messages without concern for the particular underlying connection protocol. Decisions about the best communications protocol between clients and brokers may be postponed until deployment time when the particular network requirements of an installation are known.

[0069] The transport protocol tunnel connection layer may use a novel tunneling protocol to overcome limitations of the underlying transport protocol in the messaging environment. A transport protocol may be designed as a stateless request-response mechanism, and thus may not fit the enterprise messaging system protocol model very well. For example, enterprise messaging system applications may expect asynchronous message delivery whereas a transport protocol may use a synchronous request response model. As another example, each transport protocol request-response exchange may carry a finite amount of data and is usually short lived. Thus, an enterprise messaging system message may be delivered using multiple transport protocol requests. However there may be no correlation between transport protocol requests generated by the same client. The Web servers and transport protocol intermediaries (e.g. proxy servers) thus cannot guarantee that the transport protocol requests will be processed in the same sequence as they were sent. As yet another example, TCP protocol uses a flow control mechanism to limit the network resource usage. If an enterprise messaging system message sender application generates messages very rapidly, and if each message is sent to the server using a separate transport protocol request or multiple transport protocol requests, resources on the transport protocol servers and intermediaries may be exhausted. As still yet another example, an enterprise messaging system client application may maintain a connection and a steady message exchange rate over very long periods of time (many days or even months). If the transport protocol is used, a failure on a common transport protocol proxy server may disrupt the communication between the client application and the enterprise messaging system broker.

[0070] The transport protocol tunnel connection layer may provide a connection-oriented, bi-directional byte stream service between nodes in a messaging system. Transport protocol messages may be carried as transport protocol packet payloads. The transport protocol tunnel connection layer may allow messaging system clients to access messaging system brokers using the transport protocol instead of using direct TCP connections. This enables the clients to access the brokers through firewalls. Also, with the help of a transport protocol proxy, the clients may access the messaging service even when there is no direct IP connectivity with the broker.

[0071] Using the transport protocol tunnel connection layer, if a client is separated from a broker by a firewall, messaging may be run on top of transport protocol connections, which are normally allowed through firewalls. On the client side, a transport protocol transport driver may encapsulate messages into transport protocol packets, and also may ensure that these packets are sent to the Web server in the correct sequence. The client may use a transport protocol proxy server to communicate with the broker if necessary. In one embodiment, a transport protocol tunnel servlet executing within a Web server may receive the transport protocol packets and forward the entire transport protocol packets (including the message data) to the broker. In another embodiment, the transport protocol tunnel servlet executing within the Web server may be used to pull client data (messages) out of the transport protocol packets before forwarding the data to the broker. In one embodiment, the tunnel servlet may multiplex message data from multiple clients onto one TCP connection to the broker, thus allowing the use of one tunnel servlet for multiple clients. The transport protocol tunnel servlet may also send broker data (messages) to the client in response to transport protocol pull requests. On the broker side, a transport protocol transport driver may unwrap and demultiplex incoming messages from the transport protocol tunnel servlet. The transport protocol tunnel connection layer may also be used by brokers to initiate communications with a client.

[0072] Though embodiments of the system and method are described herein as providing Hypertext Transport Protocol (HTTP) tunnel connections between entities such as clients and servers in a messaging system, it is noted that embodiments of the system and method using other unreliable/connectionless transport protocols that may not have built-in TCP support such as UDP (User Datagram Protocol), IrDA (Infrared Data Association), IBM's SNA (Systems Network Architecture), Novell's IPX (Internetwork Packet eXchange), and Bluetooth are contemplated. These embodiments may be used to provide tunnel connections between entities in messaging systems using the other transport protocols.

[0073] **FIG. 3A** illustrates a client-server messaging system implementing an HTTP tunnel connection layer according to one embodiment. Client **200** may generate messages using the messaging protocol **212**. In one embodiment, an Application Programming Interface (API) to the messaging protocol may be used by the client application to generate the messages. In one embodiment, the messaging protocol is the Java Message Service (JMS). Other messaging protocols may be used. Generated messages may then be passed to the HTTP tunnel client driver **220**.

[0074] The HTTP tunnel client driver **220** may then send the messages as HTTP POST request payloads. The HTTP tunnel client driver **220** may also use separate HTTP requests to periodically pull any data sent by the other end of the connection. The HTTP requests may be sent through HTTP proxy **206**, Internet **204**, and firewall **210** to Web server **208**. On Web server **208**, the HTTP tunnel servlet **214** may act as a transceiver and may multiplex the HTTP requests from multiple clients onto a single TCP connection **216** with the broker **202**. The HTTP tunnel broker driver **240** may receive the HTTP requests from the Web server **210** over TCP connection **216**.

[0075] Note that the HTTP proxy **206** and/or the firewall **210** are optional. In other words, the HTTP tunnel connection layer is configurable to transmit messages encapsulated in HTTP requests between entities over the Internet **104** both with and without the messages passing through proxies and/or firewalls. Also note that the Web server **208**, HTTP tunnel servlet **214**, and the broker **202** may be implemented on the same host machine or on different host machines. Note also that a Web server **208** and HTTP tunnel servlet **214** may be used to access multiple brokers **202**.

[0076] In one embodiment, the packet delivery service provided by the HTTP tunnel drivers may occasionally lose packets. Hence the HTTP tunnel connection layer may use positive acknowledgements of received packets, and may retransmit any lost packets. When a receiver receives a packet including a message, the receiver responds to the packet by sending an acknowledgement packet to the sender.

[0077] In one embodiment, the HTTP tunnel connection layer may use a sliding window protocol to implement packet sequencing and flow control on top of HTTP. In a distributed application environment using a messaging service, quite often entities may need to send a stream of packets, without errors, and with guaranteed order of delivery (i.e. that the packets are received in the order they are sent). HTTP does not guarantee packets to be received by a receiver (e.g. broker) in the same order the packets were sent by a transmitter (e.g. client). Also, HTTP does not provide a method to control the rate at which packets are sent to a receiver, thus running the risk of exhausting network resources on the receiver and losing packets. Using a sliding window protocol provides the ability to control the rate at which packets are transmitted to a receiver. The sliding window protocol may be used to guarantee that no more than a fixed number of packets are transmitted to a receiver. The fixed number of packets may be determined by a receive buffer size on the receiver. For example, a receiver may be able to receive a maximum of 100 packets from a sender. If the sender initially transmits 70 packets, the receiver may receive the packets and process 20 of them. The receiver may send a packet or packets to notify the sender that the receiver can receive 50 packets. The sender may then transmit 30 more packets. The receiver may receive the 30 packets, and in the meantime may have processed 20 more of the original 70 packets. The receiver may then transmit a packet or packets to notify the sender that the receiver can receive 60 packets (there are still 10 of the original 70 packets and the 30 new packets in the receive buffer). Thus, the sender never sends more packets to the receiver than the quantity of packets the receiver has notified the sender it can receive.

[0078] In one embodiment, when a connection is established between two entities or nodes (e.g. a server and client), each node may inform the other of how many packets it can initially receive. In a network, a node is a connection point, either a redistribution point or an end point for data transmissions. In general, a node has programmed or engineered capability to recognize, process and/or forward transmissions to other nodes. In one embodiment, each packet received by the receiver may be acknowledged with a packet sent to the sender. The acknowledgement packets may each include information indicating the current receive buffer size (i.e. the number of packets that the receiver can currently receive). Thus, the sender can determine the number of packets that it can send to the receiver without overwhelming the sender.

[0079] In one embodiment, the client **200**, broker **202** and the messaging protocol layers **212** may function similarly whether the underlying transport protocol is TCP or HTTP. In one embodiment, the messaging protocol layers **212** on both the client and the broker may use the same basic design and threading model for TCP and HTTP support. In one embodiment, an enterprise messaging system using the HTTP tunneling protocol may allow a messaging system application to exchange messages using the TCP, HTTP, SSL, or other protocol by changing appropriate configuration parameters at runtime. Thus, the application developer may not have to write any transport specific code. A client library and the broker **202** may handle the transport-specific details.

[0080] **FIG. 3B** illustrates a client-server messaging system implementing the HTTP tunnel connection layer with multiple clients accessing a broker through a Web server according to one embodiment. Both brokers **202**A and **202**B may have registered with Web server **208** that they are ready to receive connections from clients. Client **200**A may establish an HTTP tunnel connection to broker **202**A through Web server **208**. Client **200**B may establish an HTTP tunnel connection to broker **202**B through Web server **208**. At some point, either client may establish an HTTP tunnel connection with the other broker. Thus, a client may have multiple HTTP tunnel connections open to different brokers, and multiple clients may have HTTP tunnel connections open to one broker.

[0081] **FIG. 4** illustrates the architecture of a client-server messaging system implementing the HTTP tunnel connection layer according to one embodiment. Various components that are comprised in the network protocol stack for the HTTP tunnel connection layer are shown. These components lie between the application (client or broker) and the HTTP tunnel driver (client or broker). These components may include an HTTP tunnel input stream/HTTP tunnel output stream **270**, an HTTP tunnel socket **272**, and an HTTP tunnel connection **274**. In one embodiment, these components may be implemented as classes. In one embodiment, these components may be implemented as Java classes.

[0082] The HTTP tunnel drivers may send one request or receive one packet at a time. The primary responsibility of these drivers is to make sure that single packets are sent and/or received. This driver may not be aware of or be involved directly in the ordering, flow control or acknowledgement of packets.

[0083] The HTTP tunnel connection component **274** may interpret received packets. This component may be responsible for the sliding window protocol implementation. This component may be responsible for buffering the packets, may implement flow control, reordering and other aspects of the HTTP tunneling protocol. This component may be the primary component that is used to implement the end-to-end HTTP tunnel "virtual" connection as described herein.

[0084] The HTTP tunnel socket **272** and input and output streams **270** provide an interface between the application (e.g. client or broker) and the HTTP tunnel connection **274**. These components may include simple, atomic methods such as read, write and open connection methods that provide an abstract interface to the HTTP tunnel connection **274** for the applications, and thus may hide the HTTP tunnel connection implementation from the applications.

[0085] HTTP tunnel server socket **276** may be used by broker **202** to open a listening socket to the Web server **208** to allow the Web server to create a listening endpoint for the broker **202**. In doing so, broker **202** is establishing that it is ready to accept HTTP tunnel connections through Web server **208**. Thus, when client **200** desires to open an HTTP tunnel connection to broker **202**, the client may contact Web server **208** with a packet including information identifying broker **202**. Web server **208** may examine the packet and forward the packet to broker **202**, where an HTTP tunnel connection to client **200** may be established. Note that one or more HTTP packet buffers may be allocated on Web server **208** for the newly established HTTP tunnel connection. Thus, a client **200** initiates a connection, and a broker **202** accepts the connection. In one embodiment, brokers **202** cannot initiate connections to clients. Therefore, once an HTTP tunnel connection between a client **200** and a broker **202** is open (initiated by the client **200**), the client **200** may keep the connection open so that if the broker **202** has data to send to the client, there is a communications link established for the broker to send messages to the client on. In one embodiment, the broker **202** never sends HTTP packets directly to the client **200**; the packets are buffered on Web server **208**, and pulled from the Web server periodically by the client **200**.

[0086] **FIGS. 5A through 5C** are data flow diagrams illustrating the operation of a client-broker messaging system implementing the IITTP tunnel connection layer according to one embodiment. **FIG. 5A** illustrates the routing of HTTP packets between clients **200** and a broker **202** on HTTP tunnel connections **292** according to one embodiment. Each client **200** may include at least one client-side transmit buffer **222** and at least one client-side receive buffer **224**. Broker **202** may include at least one broker-side receive buffer **244** and at least one broker-side transmit buffer **242**. Each client **200** may establish an HTTP tunnel connection **292** to broker **202**, which may pass through network **294**, through Web server **208**, and over a TCP connection **296** to broker **202**. There may be a single TCP connection **296** between Web server **208**, or alternatively a TCP connection **296** may be established for each IITTP tunnel connection **292**. In passing through network **294**, an HTTP tunnel connection **292** may pass through a proxy server (not shown) and/or through one or more firewalls (not shown).

[0087] Clients **200** may generate messaging system messages, which may be buffered in a client-side transmit buffer **222**. Buffering message data may allow messages to be retransmitted if necessary, for example. The clients **200** may generate HTTP packets that include the message data as payloads and transmit the HTTP message packets to broker **202** via the HTTP tunnel connections **292** as indicated at **280A** and **280B**. Web server **208** may receive the IITTP message packets from network **294** and forward the packets to broker **202** over a TCP connection **296**.

[0088] Broker **202** may buffer incoming messages in a broker-side receive buffer **244**. Broker **202** may generate an acknowledgment (ACK) HTTP packet to acknowledge the receipt of each HTTP packet successfully received from a client **200**, and send the ACK packets to Web server **208** over a TCP connection **296** as indicated at **282A** and **282B**. Web server **208** may buffer the ACK packets received from broker **202** in buffers **250**. In one embodiment, there may be a buffer **250** for each HTTP tunnel connection **292**; in other words, each connection **292** may use a separate instance of buffer **250**. In another embodiment, one buffer **250** may be shared among two or more HTTP tunnel connections **292**.

[0089] Using the HTTP tunneling protocol layer, a broker **202** as well as clients **200** may initiate messaging system messages. Messages generated by broker **202** may be buffered in a broker-side transmit buffer **242**. Buffering message data may allow messages to be retransmitted if necessary, for example. The broker **202** may generate HTTP packets that include the message data as payloads and transmit the HTTP packets to Web server **208** over a TCP connection **296** as indicated at **288A** and **288B**. Web server **208** may buffer the HTTP message packets received from broker **202** in buffers **250**.

[0090] As indicated at **284A** and **284B**, each client **200** may send an HTTP request packet to Web server **208** to indicate that the client **200** is ready to receive HTTP packets buffered in a buffer **250** for the client **200**. In one embodiment, each client may periodically send HTTP request packets to retrieve buffered HTTP packets from Web server **208**. In one embodiment, a separate thread on a client **200** may be responsible for periodically sending the HTTP request packets.

[0091] After Web server **208** receives an HTTP request packet from a client **200** as indicated at **284**, the Web server **208** may respond by sending the requesting client **200** one or more HTTP packets currently buffered in a buffer **250** for the client **200**. The IITTP packets may include ACK packets as sent at **282** and/or HTTP message packets as sent at **288**. Upon receiving HTTP packets from the Web server **208**, a client **200** may store the received packets in a client-side receive buffer **224** to await processing. A client **200** may also generate an ACK packet and send them to broker **202** over the HTTP tunnel connection, as indicated at **290A** and **290B**, in response to each HTTP message packet received.

[0092] **FIG. 5B** illustrates the process of sending a message from a client **200** to a broker **202** via an HTTP tunnel connection according to one embodiment. At **300**, the client **200** may generate a message **260A**. At **302**, the client side HTTP tunneling driver **220** may receive the message **260A** and buffer the outgoing message data in a transmit buffer **222**, which may allow the message data to be retransmitted if necessary. The client side HTTP tunneling driver **220** generates an HTTP POST request with the message data as payload as indicated at **304**. This HTTP request may travel over the Internet. The client **200** may be configured to send HTTP requests via a proxy server **206** and through one or more firewalls **210** if necessary.

[0093] A Web server **208** may receive the HTTP request with the message data, and forward the HTTP request to the server side HTTP tunneling driver **240** over a TCP connection as indicated at **306**. The server side HTTP tunneling driver **240** may store the received message data in a receive buffer **244**. The packet header of the HTTP request may include a sequence number for use in preserving message order when multiple messages are transmitted. The message data may remain in receive buffer **244** until the broker **202** consumes it.

[0094] The server side HTTP tunneling driver may generate an acknowledgement (ACK) HTTP packet to indicate successful receipt of the HTTP request packet and message data. The ACK packet may include information about how much space is left in the receive buffer **244**. This information may be used as a "flow control" mechanism to slow down the sender (client) if the receiver (broker) cannot consume the data fast enough. The ACK packet may be sent to the Web server **208** over the TCP connection as indicated at **308**. The ACK packet may be stored in packet buffer **250A** on the Web server **208** waiting for an HTTP request from the client **200**. In one embodiment, the Web server **208** may not initiate communication with the client **200**; it can only respond to incoming HTTP requests.

[0095] The client side HTTP tunneling driver **220** then may send an HTTP request packet to the Web server **208** to pull any pending HTTP packets as indicated at **310**. In one embodiment, the client side HTTP tunneling driver **220** may use a separate reader thread that continuously sends requests to the Web server **208** to pull any pending HTTP packets. After the Web server **208** receives a pull request for the HTTP packet buffered at **308**, the Web server may send the buffered HTTP packet(s) to client **200** in response to the pull request as indicated at **312**. The client side IITTP tunneling driver **220** then may process the HTTP packet(s) including the ACK packet, and free the corresponding message data buffered in transmit buffer **222** at **302**. In one embodiment, information from the HTTP packet(s) sent to the client **200** may be stored in a client-side receive buffer (not shown) and accessed from the client-side receive buffer for processing.

[0096] **FIG. 5C** illustrates the process of sending a message from the broker **202** to the client **200** via an HTTP tunnel connection according to one embodiment. At **400**, the broker **202** generates a message **260**B. At **402**, the broker side HTTP tunneling driver **240** may receive the message **260**B and buffer the outgoing message data in a transmit buffer **242** so that it can be retransmitted if necessary. The broker side HTTP tunneling driver **240** may generate an HTTP packet with the message data as payload and forward it to Web server **208** over a TCP connection as indicated at **404**. The Web server **208** receives the HTTP packet with the message data, and writes the packet to buffer **250**. The server side HTTP tunneling driver **240** may send an HTTP request to the Web server **208** to pull any pending packets as indicated at **406**. When the Web server **208** receives the pull request, it sends the packet buffered at **404** in response to the pull request as indicated at **408**. The client side HTTP tunneling driver **220** may store the received message data in a receive buffer **224**. The packet header of the HTTP request may include a sequence number to preserve message order. The message data may remain in receive buffer **224** until the client **200** consumes it.

[0097] The client side HTTP tunneling driver generates an acknowledgement (ACK) HTTP packet to indicate successful receipt of the HTTP request and message data. The ACK packet may also include information about how much space is left in receive buffer **224**. This information may be used as a "flow control" mechanism to slow down the sender (broker) if the receiver (client) cannot consume the data fast enough. The ACK packet may be sent to the Web server **208** over the Internet as indicated at **410**. Web server **208** may forward the ACK packet to the server side HTTP tunneling driver **240** over a TCP connection as indicated at **412**. The server side HTTP tunneling driver **240** then may free the corresponding message data packet(s) buffered in transmit buffer **242** at **402**. In one embodiment, information from the ACK packet sent to the broker **202** may be stored in a broker-side receive buffer (not shown) and accessed from the broker-side receive buffer for processing.

[0098] FIGS. **6**A-**6**C are flowcharts illustrating a method of sending messages from a messaging system client to a messaging system broker over an HTTP tunnel connection layer according to one embodiment. In **FIG. 6A, a** transport protocol tunnel connection between a client and a broker in a messaging system may be established as indicated at **600**. As indicated at **602**, one or more messaging system messages may be generated on the client. In one embodiment, the generated messages may then be stored in a client-side transmit buffer. One or more transport protocol packets encapsulating the one or more messages may be generated on the client as indicated at **604**. In one embodiment, a client-side HTTP tunnel connection driver may generate the HTTP packets and include the messages as payloads of the packets. As indicated at **606**, the one or more HTTP packets may then be transmitted to the broker via the HTTP tunnel connection. In one embodiment, the client-side HTTP tunnel connection driver may handle the transmission of the packets.

[0099] In one embodiment, each HTTP packet transmitted on the HTTP tunnel connection may include message sequence information configured for use by the receiver in processing received messages in the correct sequence. This is useful since HTTP and some other transport protocols do not guarantee delivery of messages in order. In one embodiment, flow control may be applied to the sending of messages from a sender to a receiver. In flow control, before sending new messages, the sender may determine available resources on the receiver to receive new messages, and then transmit no more messages than the receiver can handle based upon the available resources. In one embodiment, the receiver may inform the sender of available space in a receive buffer to store incoming messages awaiting processing. In one embodiment, upon establishment of an HTTP tunnel connection, each side (both clients and brokers can be senders and/or receivers) may inform the other of its receive buffer size.

[0100] As indicated at **608**, the broker may receive the HTTP packets and then may acknowledge receipt of the one or more packets by sending one or more acknowledgement (ACK) packets to the client via the HTTP tunnel connection. In one embodiment, the ACK packets may each include the currently available space in the broker-side receive buffer. Thus, the sender (client) can keep track of the current receive capacity of the receiver (broker).

[0101] **FIG. 6B** expands on **606** of **FIG. 6**A and illustrates a method of transmitting one or more packets from a client to a broker via an HTTP tunnel connection according to one embodiment. As indicated at **620**, the transmitted HTTP packets may optionally pass through a proxy server. As indicated at **622**, the transmitted HTTP packets may optionally pass through one or more firewalls. A Web server may then receive the one or more HTTP packets as indicated at **624**. For example, the client may transmit the packets to a proxy server, the proxy server may transmit the packets through a firewall onto the Internet, and the packets may be received through another firewall by the Web server.

[0102] The Web server may then forward the received HTTP packets to the broker as indicated at **626**. In one embodiment, the packets may be transmitted to the broker via a TCP connection between the Web server and the broker that serves as one segment of the HTTP tunnel connection. In one embodiment, an HTTP tunnel servlet on the Web server node may serve as an interface between the Web server and the TCP connection to the broker. In one embodiment, one Web server and HTTP tunnel servlet may be used by two or more clients to communicate to a broker via HTTP tunnel connections. In this embodiment, the HTTP tunnel servlet may multiplex HTTP packets from the two or more clients onto the TCP connection. In one embodiment, the HTTP tunnel servlet may extract the messaging system message information from received HTTP packets and send only the message information to the broker over the TCP connection.

[0103] As indicated at **628**, the broker may receive the HTTP packets from the Web server. In one embodiment, the broker may receive the packets on a TCP connection to the Web server. In one embodiment, a broker-side HTTP tunnel driver may receive the packets. In one embodiment, the broker may extract the messaging system messages from the HTTP packets and store the messages in a broker-side receive buffer. In another embodiment, the entire HTTP packet may be stored in a receive buffer. In one embodiment, there may be one receive buffer on the broker for each HTTP tunnel connection. In another embodiment, a single receive buffer may be used for two or more HTTP tunnel connections.

[0104] **FIG. 6C** expands on **608** of **FIG. 6**A and illustrates a method of a broker acknowledging to a client the receipt of HTTP packets from the client via the HTTP tunnel connection according to one embodiment. As indicated at **640**, the broker may generate and send one or more acknowledgement (ACK) packets to the Web server. In one embodiment, the broker may store ACK packets in a broker-side transmit buffer. In one embodiment, the broker may send one ACK packet for each received HTTP packet. In another embodiment, the broker may send one ACK packet for each completely received messaging system message. In one embodiment, the ACK packets may be sent to the Web server over a TCP connection. In one embodiment, a broker-side HTTP tunnel driver may handle the transmission of the ACK packets on the TCP connection. In one embodiment, an HTTP tunnel servlet may receive the packets for the Web server.

[0105] As indicated at **642**, the Web server may store the acknowledgement packets in an HTTP packet buffer. In one embodiment, there may be one HTTP packet buffer for each HTTP tunnel connection supported by the Web server. In another embodiment, there may be two HTTP packet buffers per tunnel connection, with one HTTP packet buffer for each message flow direction on each tunnel connection. In yet another embodiment, one or more HTTP packet buffers may be used for two or more tunnel connections.

[0106] As indicated at **644**, at some point, the client may send an HTTP pull request packet to the Web server. In one embodiment, the client may periodically send HTTP pull requests to the Web server. In one embodiment, a separate thread on the client may handle periodically sending pull requests. The Web server may transmit to the client the one or more HTTP ACK packets stored in the HTTP packet buffer associated with the client in response to receiving the pull request packet as indicated at **646**. As indicated at **648**, the transmitted ACK packets may optionally pass through one or more firewalls. As indicated at **750**, the transmitted ACK packets may optionally pass through a proxy server. As indicated at **652**, the client may then receive the one or more ACK packets. In one embodiment, each ACK packet may include information on available space in the broker-side receive buffer to be used in flow control of HTTP packets from the client to the broker. The ACK packets may serve to acknowledge the receipt of the transmitted data so that the sender may free its transmit buffers.

[0107] FIGS. **7**A-7C are flowcharts illustrating a method of sending messages from a messaging system client to a messaging system broker over an HTTP tunnel connection layer according to one embodiment. In **FIG. 7A, a** transport protocol tunnel connection between a client and a broker in a messaging system may be established as indicated at **700**. As indicated at **702**, one or more messaging system messages may be generated on the broker. In one embodiment, the generated messages may then be stored in a broker-side transmit buffer. One or more transport protocol packets encapsulating the one or more messages may be generated on the broker as indicated at **704**. In one embodiment, a broker-side HTTP tunnel connection driver may generate the HTTP packets and include the messages as payloads of the packets. As indicated at **706**, the one or more HTTP packets may then be transmitted to the client via the HTTP tunnel connection. In one embodiment, the broker-side HTTP tunnel connection driver may handle the transmission of the packets.

[0108] As indicated at **708**, the client may receive the HTTP packets, and then may acknowledge receipt of the one or more packets by sending one or more acknowledgement (ACK) packets to the broker via the HTTP tunnel connection. In one embodiment, the ACK packets may each include the currently available space in the client-side receive buffer. Thus, the sender (broker) can keep track of the current receive capacity of the receiver (client).

[0109] **FIG. 7B** expands on **706** of **FIG. 7**A and illustrates a method of transmitting one or more packets from a broker to a client via an HTTP tunnel connection according to one embodiment. As indicated at **720**, the broker may generate and send one or more HTTP packets to the Web server. In one embodiment, the broker may store the HTTP packets in a broker-side transmit buffer. In one embodiment, the HTTP packets may be sent to the Web server over a TCP connection. In one embodiment, a broker-side HTTP tunnel driver may handle the transmission of the HTTP packets on the TCP connection.

[0110] As indicated at **722**, the Web server may receive the HTTP packets and store the received HTTP packets in an HTTP packet buffer. In one embodiment, an HTTP tunnel servlet may receive the packets for the Web server. As indicated at **724**, at some point, the client may send an HTTP pull request packet to the Web server. In one embodiment, the client may periodically send HTTP pull requests to the Web server. The Web server may transmit to the client the one or more HTTP packets stored in the HTTP packet buffer associated with the client in response to receiving the pull request packet as indicated at **726**. As indicated at **728**, the transmitted HTTP packets may optionally pass through one or more firewalls. As indicated at **730**, the transmitted HTTP packets may optionally pass through a proxy server. As indicated at **732**, the client may then receive the one or more HTTP packets. In one embodiment, the client may store the received HTTP packets in a client-side receive buffer. In one embodiment, each HTTP packet may include sequence information configured for use by the client in processing received messages in sequence.

[0111] **FIG. 7C** expands on **708** of **FIG. 7**A and illustrates a method of a client acknowledging to a broker the receipt of HTTP packets from the broker via the HTTP tunnel connection according to one embodiment. As indicated at **740**, the client may generate and send one or more acknowledgement (ACK) packets to the broker. As indicated at **742**, the transmitted ACK packets may optionally pass through a proxy server. As indicated at **744**, the transmitted ACK packets may optionally pass through one or more firewalls. A Web server may then receive the one or more ACK packets as indicated at **746**. For example, the client may transmit the ACK packets to a proxy server, the proxy server may transmit the ACK packets through a firewall onto the Internet, and the ACK packets may be received through another firewall by the Web server.

[0112] The Web server may then forward the received ACK packets to the broker as indicated at **748**. In one embodiment, the ACK packets may be transmitted to the broker via a TCP connection between the Web server and the broker that serves as one segment of the HTTP tunnel connection. In one embodiment, an HTTP tunnel servlet on the Web server node may serve as an interface between the Web server and the TCP connection to the broker.

[**0113**] As indicated at **750**, the broker may receive the ACK packets from the Web server. In one embodiment, the broker may receive the packets on a TCP connection to the Web server. In one embodiment, a broker-side HTTP tunnel driver may receive the ACK packets. In one embodiment, the received ACK packets may be stored in a broker-side receive buffer. In one embodiment, each ACK packet may include information on available space in the client-side receive buffer to be used in flow control of HTTP packets from the broker to the client. The methods as described in FIGS. **6A-6C** and FIGS. **7A-7C** may be implemented in software, hardware, or a combination thereof. The order of method may be changed, and various steps may be added, reordered, combined, omitted, modified, etc.

[**0114**] **FIG. 8** illustrates an exemplary HTTP tunneling protocol packet format according to one embodiment. The HTTP tunnel drivers at either end of a connection may encode all their data packets using this packet format. Each messaging system message may be carried separately as a HTTP request or response payload. A messaging system message may be sent as the payload of a single packet or, alternatively, the message may be broken into parts and sent as the payload of two or more packets.

[**0115**] The packet format may include several fields. Version **500** may indicate a version number of the packet that may be used to indicate different releases of the HTTP tunnel connection layer software. In one embodiment, the version **500** may be a 16-bit field. Packet type **502** may indicate the type of the packet. This field may be used to indicate to the HTTP tunnel driver at the other end what to do with the contents of this packet. In one embodiment, the packet type field may be 16 bits. Size **504** may indicate the size of the entire packet including the header. In one embodiment, this may be a 32-bit field. Connection ID **506** may be a unique integer that may be used as connection identifier. This value may be assigned at the time of connection establishment. This field may be used by the server to distinguish between connections to multiple clients. In one embodiment, this may be a 32-bit field. Packet sequence number **508** may be used to ensure sequential delivery of data bytes and flow control. In one embodiment, each packet may be assigned a unique incremental sequence number by the sender. In one embodiment, this may be a 32-bit field. One embodiment may include a receiver window size **510** that may be used for flow control and may indicate the capacity of the receive side buffer. In one embodiment, this may be a 16-bit field. One embodiment may guarantee that the packets are either delivered correctly, or in case of an error, are not delivered at all. Some embodiments may not use checksum **512**, and thus this field may be reserved.

[**0116**] One embodiment of the HTTP tunnel connection layer may use a connection establishment protocol that may be used to initialize the following connection state components:

[**0117**] The HTTP tunnel servlet **214** allocates a unique connection ID for the new connection. It also may allocate one or more buffers **250** for packet streams in both directions.

[**0118**] The HTTP tunnel drivers on both server and client side allocate and initialize the transmit buffers and receive buffers for packet streams in both directions.

[**0119**] The following are examples of HTTP tunneling protocol packets that may be used in the connection establishment protocol and are not intended to be limiting. Note that one skilled in the art will recognize that other packet formats may be used within the scope of the invention.

[**0120**] A client **200** may initiate a connection to a server **202** by sending the following information in an HTTP packet to the HTTP tunnel servlet **214**:

[**0121**] URL parameters="?ServerName=<ServerIdString>&Type=connect" Packet Type=connection initialization packet (e.g. CONN_INIT_PACKET (1)).

[**0122**] Connection ID =0

[**0123**] The servlet **214** may allocate a unique connection ID for this connection and send it to the client **200** and the server **202** in HTTP packets including the following information:

[**0124**] Packet Type=connection initialization packet (e.g. CONN_INIT_PACKET (1)).

[**0125**] ConnectionID=<connid>

[**0126**] This information may be sent to the client **200** as a response payload for the HTTP request that carried the client's connection initialization packet.

[**0127**] The server **202** may acknowledge the connection initialization packet to the client **200** in an ACK packet that may include the following information:

[**0128**] Pull URL parameters=

[**0129**] "?ServerName=<ServerIdString>&Type=pull&ConnId=<connid>"

[**0130**] Packet Type=

[**0131**] connection initialization acknowledgement (e.g. CONN_INIT_ACK (2)).

[**0132**] After completion, both client **200** and server **202** are aware of the newly established connection and normal data exchange can begin.

[**0133**] Once the connection is established, both sides may start sending data and connection management packets. The following are examples of HTTP tunneling protocol packets that may be used as data and connection management packets and are not intended to be limiting. Note that one skilled in the art will recognize that other packet formats may be used within the scope of the invention.

[**0134**] From the client **200** to the HTTP tunnel servlet **214** to the server **202**:

[**0135**] URL parameters="?ServerName=<ServerIdString>&Type=push"

[**0136**] From the server **202** to the HTTP tunnel servlet **214** to the client **200**:

[**0137**] Pull URL parameters=

[**0138**] "?ServerName=<ServerIdString>&Type=pull&ConnId=<connid>"

[0139] Each outgoing data packet (e.g. Packet Type=DATA_PACKET) may be assigned an incremental sequence number by the sender. The receiver may check the packet sequence number with its receive window. Any duplicate packets may be discarded.

[0140] After consuming a packet, the receiver may acknowledge the highest contiguous sequence number by sending the ACK packet as follows:

[0141] Packet Type=acknowledgement (e.g. ACK)

[0142] Connection ID=<connid>

[0143] Sequence=<sequence number of the data packet being acknowledged>

[0144] Receive Window Size=<remaining receive window capacity>

[0145] When an acknowledgement is received, the sender may update the round trip time for the connection. In one embodiment, a simple linear function of the computed round trip time may be used as the packet retransmission interval.

[0146] In one embodiment, when an acknowledgement packet reports a "Receive Window Size" of zero, the sender may stop sending further packets. The sender may send periodic repeat transmissions of the next packet to force the receiver to send a window update as soon as it is ready to receive more data. When the receiver indicates that it is ready to receive more data, the sender may resume sending packets.

[0147] In one embodiment, the HTTP Tunneling protocol may support the following runtime connection option. The client pull period is the duration (e.g. in seconds) of the idle period between consecutive pull requests sent by the client. If this value is positive, whenever the HTTP tunnel driver on the client side receives an empty response to its Pull request, the driver may sleep for the specified period before issuing another pull request. This may help conserve the servlet **214**'s resources and hence improve connection scalability.

[0148] Either client **200** or server **202** may set connection options. This may be used to control the runtime parameters associated with a connection. For example, this may be used to control how often packets are pulled from a Web server. As another example, this may be used to configure how long a client may remain inactive before the connection is dropped. The connection option values are part of the connection state information, and hence, in one embodiment, an HTTP packet including the following information may be used to update the connection options:

[0149] Packet Type=connection option packet (e.g. CONN_OPTION_PACKET)

$$\text{Data} = \begin{array}{l}\text{Option Type (integer)}\\ \text{Option Value (integer)}\end{array}$$

[0150] In one embodiment, either end (client or server) may initiate connection shutdown by sending a connection close packet (e.g. Packet Type CONN_CLOSE_PACKET). Until both parties complete the connection shutdown, this packet may otherwise be treated like a normal data packet. This may help to ensure that all the data packets that are in the pipeline ahead of the connection close packet are processed before the connection resources are destroyed. The remote end may consume all the data and acknowledge the connection close packet, at which point the connection is terminated and all the resources may be freed.

[0151] In one embodiment, a connection abort packet may be generated by the servlet when it realizes that either the server **202** or the client **200** has terminated ungracefully. The server termination may result in an IOException on the TCP connection between the servlet **214** and the server **202**, and hence may be detected. In one embodiment, detecting client **200** termination may be handled as follows. If the servlet **214** does not receive a "pull" request from client **200** for a certain period of time, the servlet **214** may assume that the client **200** is not responding and thus may be down. The ungraceful connection shutdown may be achieved by sending a single packet with Packet Type connection abort packet (e.g. CONN_ABORT_PACKET) to the server **202** and possibly to the client **200** as well.

[0152] The following describes the initialization of the link between server **200** and HTTP tunneling servlet **214** according to one embodiment. The HTTP tunneling servlet **214** may listen on a fixed port number for TCP connections from servers. After the TCP connection is established to server **200**, the server **200** may send the following information in an HTTP tunneling protocol packet to the servlet **214**:

[0153] Packet Type: link initialization packet (e.g. LINK_INIT_PACKET)

$$\text{Data} = \begin{array}{c}\mathit{ServerIdString}\ \text{(String)}\\ \mathit{ConnectionCount}\ \text{(Integer)}\\ \text{Sequence of }\{\mathit{ConnectionID}\ \text{(integer)},\\ \mathit{ConnectionPullPeriod}\ \text{(integer)}\}\end{array}$$

[0154] The ServerIdString may be used to establish the identity of the server **200**. Clients may use this string to specify which server they want to talk to. This allows the use of a single servlet **214** as a gateway for multiple servers. The data portion of this packet may include information about any existing HTTP Tunnel connections. This allows HTTP tunnel connections to survive unexpected Web server/servlet engine failures or restarts.

[0155] Various embodiments may further include receiving, sending or storing instructions and/or data implemented in accordance with the foregoing description upon a carrier medium. Generally speaking, a carrier medium may include storage media or memory media such as magnetic or optical media, e.g., disk or CD-ROM, volatile or non-volatile media such as RAM (e.g. SDRAM, DDR SDRAM, RDRAM, SRAM, etc.), ROM, etc. as well as transmission media or signals such as electrical, electromagnetic, or digital signals, conveyed via a communication medium such as network and/or a wireless link.

[0156] In summary, a system and method for providing HTTP tunnel connections between entities such as clients and servers in a messaging system have been disclosed. It will be appreciated by those of ordinary skill having the benefit of this disclosure that the illustrative embodiments described above are capable of numerous variations without departing from the scope and spirit of the invention. Various modifications and changes may be made as would be obvious to a person skilled in the art having the benefit of this disclosure. It is intended that the following claims be interpreted to embrace all such modifications and changes and, accordingly, the specifications and drawings are to be regarded in an illustrative rather than a restrictive sense.

What is claimed is:

1. A method comprising:

establishing a transport protocol tunnel connection from a first node in a messaging system to a second node in the messaging system;

generating a messaging system message on the first node;

generating one or more transport protocol packets, wherein the one or more transport protocol packets each includes at least a part of the messaging system message; and

transmitting the one or more transport protocol packets to the second node via the transport protocol tunnel connection;

wherein the transport protocol tunnel connection provides full-duplex transmission of messaging system messages between the first node and the second node, and wherein the transport protocol tunnel connection further provides delivery of the messaging system messages in the sequence in which the messaging system messages are generated.

2. The method as recited in claim 1, further comprising storing the messaging system message in a transmit buffer on the first node after said generating the messaging system message on the first node.

3. The method as recited in claim 1, wherein the transport protocol tunnel connection passes through a proxy server.

4. The method as recited in claim 3, wherein said transmitting the one or more transport protocol packets to the second node via the transport protocol tunnel connection comprises:

transmitting the one or more transport protocol packets from the first node to the proxy server; and

transmitting the one or more transport protocol packets from the proxy server to the second node.

5. The method as recited in claim 1, wherein the transport protocol tunnel connection passes through at least one firewall.

6. The method as recited in claim 1, wherein the transport protocol tunnel connection is established through a network.

7. The method as recited in claim 6, wherein the network is the Internet.

8. The method as recited in claim 1, wherein the first node is a client in the messaging system, and wherein the second node is a broker in the messaging system.

9. The method as recited in claim 1, wherein the transport protocol tunnel connection passes through a third node, and wherein, in said transmitting the one or more transport protocol packets to the second node, the method further comprises:

transmitting the one or more transport protocol packets to the third node; and

the third node forwarding the one or more transport protocol packets to the second node.

10. The method as recited in claim 9, wherein the one or more transport protocol packets are forwarded to the second node via a Transmission Control Protocol (TCP) connection portion of the transport protocol tunnel connection between the third node and the second node.

11. The method as recited in claim 9, wherein the third node is a Web server.

12. The method as recited in claim 1, wherein the transport protocol tunnel connection passes through a proxy server and a Web server, and wherein said transmitting the one or more transport protocol packets to the second node via the transport protocol tunnel connection comprises:

transmitting the one or more transport protocol packets from the first node to the proxy server;

transmitting the one or more transport protocol packets from the proxy server to the Web server; and

the Web server forwarding the one or more transport protocol packets to the second node.

13. The method as recited in claim 12, wherein the transport protocol tunnel connection passes through at least one firewall between the proxy server and the Web server.

14. The method as recited in claim 1, wherein the one or more transport protocol packets include messaging system message sequence information configured for use in processing two or more messaging system messages in sequence.

15. The method as recited in claim 1, further comprising:

receiving the transmitted one or more transport protocol packets on the second node; and

storing the messaging system message from the one or more transport protocol packets in a receive buffer on the second node.

16. The method as recited in claim 1, further comprising:

receiving the transmitted one or more transport protocol packets on the second node;

the second node generating an acknowledgement transport protocol packet to indicate successful receipt of the one or more transport protocol packets including the messaging system message; and

transmitting the acknowledgement transport protocol packet to the first node via the transport protocol tunnel connection.

17. The method as recited in claim 16, further comprising:

storing the messaging system message from the received one or more transport protocol packets in a receive buffer on the second node;

wherein the acknowledgement transport protocol packet includes information indicating available space in the receive buffer, and wherein the information indicating available space in the receive buffer is configured for use in flow control of messaging system messages transmitted from the first node to the second node.

18. The method as recited in claim 17, further comprising:

receiving the transmitted acknowledgement transport protocol packet on the first node;

generating one or more messaging system messages on the first node;

storing the one or more messaging system messages in a transmit buffer on the first node;

determining from the information indicating available space in the receive buffer included in the received acknowledgement transport protocol packet if there is space available to receive the one or more messaging system messages on the second node;

if said determining indicates there is space available to store the one or more messaging system messages in the receive buffer of the second node:

generating a second one or more transport protocol packets, wherein the second one or more transport protocol packets include the one or more messaging system messages; and

transmitting the second one or more transport protocol packets to the second node via the transport protocol tunnel connection; and

if said determining indicates there is not space available to store the second messaging system message in the receive buffer of the second node, inhibiting generating the second one or more transport protocol packets including the one or more messaging system messages.

19. The method as recited in claim 18, further comprising:

the first node receiving a transport protocol packet indicating available space in the receive buffer of the second node;

determining from the information indicating available space in the receive buffer included in the received transport protocol packet that there is space available to receive the one or more messaging system messages on the second node;

generating the second one or more transport protocol packets, wherein the second one or more transport protocol packets include the one or more messaging system messages; and

transmitting the second one or more transport protocol packets to the second node via the transport protocol tunnel connection.

20. The method as recited in claim 16, wherein the transport protocol tunnel connection passes through a third node, and wherein, in said transmitting the acknowledgement transport protocol packet to the first node, the method further comprises:

transmitting the acknowledgement transport protocol packet to the third node; and

storing the acknowledgement transport protocol packet in a transport protocol packet buffer on the third node.

21. The method as recited in claim 20, wherein, in said transmitting the acknowledgement transport protocol packet to the first node, the method further comprises:

the first node transmitting a transport protocol request packet to the third node; and

the third node transmitting the acknowledgement transport protocol packet stored in the transport protocol packet buffer to the first node via the transport protocol tunnel connection in response to the transport protocol request packet.

22. The method as recited in claim 21, wherein the acknowledgement transport protocol packet is transmitted to the third node via a Transmission Control Protocol (TCP) connection portion of the transport protocol tunnel connection.

23. The method as recited in claim 16, wherein the transport protocol tunnel connection passes through a third node, and wherein, in said transmitting the acknowledgement transport protocol packet to the first node, the method further comprises:

transmitting the acknowledgement transport protocol packet to the third node; and

the third node forwarding the acknowledgement transport protocol packet to the first node.

24. The method as recited in claim 23, wherein the acknowledgement transport protocol packet are forwarded to the first node via a Transmission Control Protocol (TCP) connection portion of the transport protocol tunnel connection.

25. The method as recited in claim 1, wherein the first node is a server in the messaging system, and wherein the second node is a client in the messaging system.

26. The method as recited in claim 1, wherein the transport protocol tunnel connection passes through a third node, and wherein, in said transmitting the one or more transport protocol packets to the second node via the transport protocol tunnel connection, the method further comprises:

transmitting the one or more transport protocol packets to the third node; and

storing the one or more transport protocol packets in a transport protocol packet buffer on the third node.

27. The method as recited in claim 26, wherein, in said transmitting the one or more transport protocol packets to the second node via the transport protocol tunnel connection, the method further comprises:

the second node sending one or more transport protocol request packets to the third node; and

the third node transmitting the one or more transport protocol packets stored in the transport protocol packet buffer to the second node via the transport protocol tunnel connection in response to the one or more transport protocol request packets.

28. The method as recited in claim 26, wherein the third node is a Web server.

29. The method as recited in claim 1, wherein the transport protocol is Hypertext Transport Protocol (HTTP).

30. The method as recited in claim 1, wherein the transport protocol is one of UDP (User Datagram Protocol), IrDA (Infrared Data Association), SNA (Systems Network Architecture), IPX (Internetwork Packet eXchange), and Bluetooth.

31. A method comprising:

establishing a Hypertext Transport Protocol (HTTP) tunnel connection from a first node in a messaging system to a second node in the messaging system;

generating a messaging system message on the first node;

generating one or more HTTP packets, wherein the one or more HTTP packets each includes at least a part of the messaging system message; and

transmitting the one or more HTTP packets to the second node via the HTTP tunnel connection;

wherein the HTTP tunnel connection provides full-duplex transmission of messaging system messages between the first node and the second node, and wherein the HTTP tunnel connection further provides delivery of the messaging system messages in the sequence in which the messaging system messages are generated.

32. The method as recited in claim 31, wherein the first node is a client in the messaging system, wherein the HTTP tunnel connection passes through a proxy server, and wherein said transmitting the one or more HTTP packets to the second node via the HTTP tunnel connection comprises:

transmitting the one or more HTTP packets from the client to the proxy server; and

transmitting the one or more HTTP packets from the proxy server to the second node.

33. The method as recited in claim 31, wherein the HTTP tunnel connection is established through the Internet, and wherein the HTTP tunnel connection passes through at least one firewall.

34. The method as recited in claim 31, wherein the first node is a client in the messaging system, and wherein the second node is a broker in the messaging system.

35. The method as recited in claim 31, wherein the HTTP tunnel connection passes to through a Web server, wherein the second node is a broker in the messaging system, and wherein, in said transmitting the one or more HTTP packets to the second node, the method further comprises:

transmitting the one or more HTTP packets to the Web server; and

the Web server forwarding the one or more HTTP packets to the broker via a Transmission Control Protocol (TCP) connection portion of the HTTP tunnel connection between the Web server and the broker.

36. The method as recited in claim 31, wherein the first node is a client in the messaging system, wherein the second node is a broker in the messaging system, wherein the HTTP tunnel connection passes through a proxy server and a Web server, and wherein said transmitting the one or more HTTP packets to the broker via the HTTP tunnel connection comprises:

transmitting the one or more HTTP packets from the client to the proxy server;

transmitting the one or more HTTP packets from the proxy server to the Web server; and

the Web server forwarding the one or more HTTP packets to the broker;

wherein the HTTP tunnel connection passes through at least one firewall between the proxy server and the Web server.

37. The method as recited in claim 31, wherein the one or more HTTP packets include messaging system message sequence information configured for use in processing two or more messaging system messages in sequence.

38. The method as recited in claim 31, further comprising:

receiving the transmitted one or more HTTP packets on the second node;

storing the messaging system message from the one or more HTTP packets in a receive buffer on the second node;

the second node generating an acknowledgement HTTP packet to indicate successful receipt of the one or more HTTP packets including the messaging system message; and

transmitting the acknowledgement HTTP packet to the first node via the HTTP tunnel connection.

39. The method as recited in claim 38, wherein the acknowledgement HTTP packet includes information indicating available space in the receive buffer, the method further comprising:

receiving the transmitted acknowledgement HTTP packet on the first node;

generating one or more messaging system messages on the first node;

storing the one or more messaging system messages in a transmit buffer on the first node;

determining from the information indicating available space in the receive buffer included in the received acknowledgement HTTP packet that there is not space available to receive the one or more messaging system messages on the second node;

the first node receiving an HTTP packet from the second node indicating available space in the receive buffer of the second node;

determining from the information indicating available space in the receive buffer included in the received HTTP packet that there is space available to receive the one or more messaging system messages on the second node;

generating a second one or more HTTP packets, wherein the second one or more HTTP packets include the one or more messaging system messages; and

transmitting the second one or more HTTP packets to the second node via the HTTP tunnel connection.

40. The method as recited in claim 38, wherein the first node is a client in the messaging system, wherein the HTTP tunnel connection passes through a Web server, and wherein, in said transmitting the acknowledgement HTTP packet to the first node, the method further comprises:

transmitting the acknowledgement HTTP packet to the Web server;

storing the acknowledgement HTTP packet in an HTTP packet buffer on the Web server;

the client sending an HTTP request packet to the Web server; and

the Web server transmitting the acknowledgement HTTP packet stored in the HTTP packet buffer to the client via the HTTP tunnel connection in response to the HTTP request packet.

41. The method as recited in claim 38, wherein the first node is a broker in the messaging system, wherein the HTTP tunnel connection passes through a Web server, and wherein, in said transmitting the acknowledgement HTTP packet to the first node, the method further comprises:

transmitting the acknowledgement IITTP packet to the Web server; and

the Web server forwarding the acknowledgement HTTP packet to the first node via a Transmission Control Protocol (TCP) connection portion of the HTTP tunnel connection.

42. The method as recited in claim 31, wherein the first node is a server in the messaging system, and wherein the second node is a client in the messaging system.

43. The method as recited in claim 31, wherein the second node is a client in the messaging system, wherein the IITTP tunnel connection passes through a Web server, and wherein, in said transmitting the one or more HTTP packets to the second node via the HTTP tunnel connection, the method further comprises:

transmitting the one or more HTTP packets to the Web server;

storing the one or more HTTP packets in an HTTP packet buffer on the Web server;

the client sending one or more HTTP request packets to the Web server; and

the Web server transmitting the one or more HTTP packets stored in the HTTP packet buffer to the client via the HTTP tunnel connection in response to the one or more HTTP request packets.

44. A method comprising:

establishing a transport protocol tunnel connection from a first node in a messaging system to a second node in the messaging system;

generating a sequence of messaging system messages on the first node;

generating a plurality of transport protocol packets on the first node, wherein each of the transport protocol packets includes at least a part of one of the sequence of messaging system messages, and wherein each of the transport protocol packets includes sequence information for the particular messaging system message;

transmitting the plurality of transport protocol packets to the second node in the messaging system via the transport protocol tunnel connection;

receiving the plurality of transport protocol packets on the second node; and

processing the sequence of messaging system messages on the second node, wherein said processing uses the sequence information for the plurality of messaging system messages in the plurality of transport protocol packets.

45. The method as recited in claim 44, wherein the transport protocol tunnel connection is established through the Internet, and wherein the transport protocol tunnel connection passes through at least one firewall.

46. The method as recited in claim 44, wherein the first node is a client in the messaging system, wherein the second node is a broker in the messaging system.

47. The method as recited in claim 44, wherein the second node is a broker in the messaging system, wherein the transport protocol tunnel connection passes through a Web server, and wherein said transmitting the plurality of transport protocol packets to the second node in the messaging system via the transport protocol tunnel connection comprises:

transmitting the plurality of transport protocol packets from the first node to the Web server; and

the Web server forwarding the plurality of transport protocol packets to the broker;

wherein the transport protocol tunnel connection passes through at least one firewall between the proxy server and the Web server.

48. The method as recited in claim 44, wherein the first node is a broker in the messaging system, wherein the second node is a client in the messaging system.

49. The method as recited in claim 44, wherein the second node is a client in the messaging system, wherein the transport protocol tunnel connection passes through a Web server, and wherein, in said transmitting the plurality of transport protocol packets to the second node in the messaging system via the transport protocol tunnel connection comprises:

transmitting the plurality of transport protocol packets to the Web server;

storing the plurality of transport protocol packets in a transport protocol packet buffer on the Web server;

the client sending one or more transport protocol request packets to the Web server; and

the Web server transmitting the plurality of transport protocol packets stored in the transport protocol packet buffer to the client via the transport protocol tunnel connection in response to the one or more transport protocol request packets.

50. The method as recited in claim 44, further comprising:

storing the sequence of messaging system messages from the received transport protocol packets in a receive buffer on the second node;

the second node generating an acknowledgement transport protocol packet for each of the received transport protocol packets to indicate successful receipt of the transport protocol packets including the sequence of messaging system messages; and

transmitting the acknowledgement transport protocol packets to the first node via the transport protocol tunnel connection;

wherein each of the acknowledgement transport protocol packets includes information indicating available space in the receive buffer, wherein the information indicating available space in the receive buffer is configured for use in flow control of messaging system messages transmitted from the first node to the second node.

51. The method as recited in claim 44, wherein the transport protocol is Hypertext Transport Protocol (HTTP).

52. A method comprising:

establishing a transport protocol tunnel connection from a first node in a messaging system to a second node in the messaging system;

the first node receiving a first transport protocol packet from the second node indicating available space in a receive buffer of the second node;

generating one or more messaging system messages on the first node;

storing the one or more messaging system messages in a transmit buffer on the first node;

determining from the information indicating available space in the receive buffer included in the received acknowledgement HTTP packet that there is not space available to receive the one or more messaging system messages on the second node;

the first node receiving a second transport protocol packet from the second node indicating available space in the receive buffer of the second node;

determining from the information indicating available space in the receive buffer included in the received second transport protocol packet that there is space available to receive the one or more messaging system messages on the second node;

generating one or more transport protocol packets, wherein the second one or more transport protocol packets include the generated one or more messaging system messages; and

transmitting the one or more transport protocol packets to the second node via the transport protocol tunnel connection.

53. The method as recited in claim 52, wherein the transport protocol tunnel connection is established through the Internet, and wherein the transport protocol tunnel connection passes through at least one firewall.

54. The method as recited in claim 52, wherein the first node is a client in the messaging system, wherein the second node is a broker in the messaging system.

55. The method as recited in claim 52, wherein the first node is a broker in the messaging system, wherein the second node is a client in the messaging system.

56. The method as recited in claim 52, further comprising:

receiving the one or more transport protocol packets on the second node;

storing the one or more messaging system messages from the received one or more transport protocol packets in the receive buffer on the second node;

the second node generating one or more acknowledgement transport protocol packets to indicate successful receipt of the one or more transport protocol packets including the one or more of messaging system messages; and

transmitting the one or more acknowledgement transport protocol packets to the first node via the transport protocol tunnel connection;

wherein each of the acknowledgement transport protocol packets includes information indicating available space in the receive buffer.

57. The method as recited in claim 52, wherein the one or more transport protocol packets include messaging system message sequence information configured for use in processing two or more messaging system messages in sequence.

58. The method as recited in claim 52, wherein the transport protocol is Hypertext Transport Protocol (HTTP).

59. A messaging system comprising:

a first node comprising a first memory;

a second node comprising a second memory;

wherein the first memory comprises first program instructions executable within the first node to:

establish a transport protocol tunnel connection from the first node to the second node through a network;

generate a messaging system message;

generate one or more transport protocol packets, wherein the one or more transport protocol packets each includes at least a part of the messaging system message; and

transmit the one or more transport protocol packets to the second node via the transport protocol tunnel connection;

wherein the transport protocol tunnel connection provides full-duplex transmission of messaging system messages between the first node and the second node, and wherein the transport protocol tunnel connection further provides delivery of the messaging system messages in the sequence in which the messaging system messages are generated.

60. The messaging system as recited in claim 59, wherein the first node further comprises a transmit buffer, wherein the first program instructions are further executable within the first node to store the messaging system message in the transmit buffer on the first node after said generating the messaging system message.

61. The messaging system as recited in claim 59, wherein the transport protocol tunnel connection passes through a proxy server, and wherein, in said transmitting the one or more transport protocol packets to the second node via the transport protocol tunnel connection, the first program instructions are further executable within the first node to transmit the one or more transport protocol packets from the first node to the proxy server, wherein the proxy server is configured to transmit the one or more transport protocol packets to the second node.

62. The messaging system as recited in claim 59, wherein the transport protocol tunnel connection passes through at least one firewall.

63. The messaging system as recited in claim 59, wherein the transport protocol tunnel connection is established through the Internet.

64. The messaging system as recited in claim 59, wherein the first node is a client in the messaging system, and wherein the second node is a broker in the messaging system.

65. The messaging system as recited in claim 59, wherein the messaging system further comprises:

a third node comprising a third memory;

wherein the transport protocol tunnel connection passes through the third node, and wherein, in said transmitting the one or more transport protocol packets to the second node, the first program instructions are further executable within the first node to:

transmit the one or more transport protocol packets to the third node;

wherein the third memory comprises third program instructions executable within the third node to:

receive the one or more transport protocol packets from the first node; and

forward the one or more received transport protocol packets to the second node.

66. The messaging system as recited in claim 65, wherein the one or more transport protocol packets are forwarded from the third node to the second node via a Transmission Control Protocol (TCP) connection portion of the transport protocol tunnel connection between the third node and the second node.

67. The messaging system as recited in claim 66, wherein the transport protocol tunnel connection passes through at least one firewall between the first node and the third node.

68. The messaging system as recited in claim 59, wherein the one or more transport protocol packets include messaging system message sequence information configured for use in processing two or more messaging system messages in sequence.

69. The messaging system as recited in claim 59, wherein the second memory comprises second program instructions executable within the second node to:

receive the transmitted one or more transport protocol packets;

generate an acknowledgement transport protocol packet to indicate successful receipt of the one or more transport protocol packets including the messaging system message; and

transmit the acknowledgement transport protocol packet to the first node via the transport protocol tunnel connection.

70. The messaging system as recited in claim 69, wherein the second node further comprises a receive buffer, wherein the second program instructions are further executable within the second node to:

store the messaging system message from the received one or more transport protocol packets in the receive buffer of the second node;

wherein the acknowledgement transport protocol packet includes information indicating available space in the receive buffer of the second node.

71. The messaging system as recited in claim 70, wherein the first node further comprises a transmit buffer, wherein the first program instructions are further executable within the first node to:

receive the transmitted acknowledgement transport protocol packet;

generate one or more messaging system messages;

store the one or more messaging system messages in the transmit buffer on the first node;

from the information indicating available space in the receive buffer included in the received acknowledgement transport protocol packet, determine if there is space available to receive the one or more messaging system messages on the second node;

if said determining indicates there is space available to store the one or more messaging system messages in the receive buffer of the second node:

generate a second one or more transport protocol packets, wherein the second one or more transport protocol packets include the one or more messaging system messages; and

transmit the second one or more transport protocol packets to the second node via the transport protocol tunnel connection; and

if said determining indicates there is not space available to store the second messaging system message in the receive buffer of the second node, inhibit generating the second one or more transport protocol packets including the one or more messaging system messages.

72. The messaging system as recited in claim 71, wherein the first program instructions are further executable within the first node to:

receive a transport protocol packet indicating available space in the receive buffer of the second node;

from the information indicating available space in the receive buffer included in the received transport protocol packet, determine that there is space available to receive the one or more messaging system messages on the second node;

generate the second one or more transport protocol packets, wherein the second one or more transport protocol packets include the one or more messaging system messages; and

transmit the second one or more transport protocol packets to the second node via the transport protocol tunnel connection.

73. The messaging system as recited in claim 69, further comprising:

a third node comprising:

a third memory; and

a transport protocol packet buffer;

wherein the transport protocol tunnel connection passes through the third node, wherein the third memory comprises third program instructions executable within the third node to:

receive the acknowledgement transport protocol packet transmitted to the first node via the transport protocol tunnel connection from the second node; and

store the received acknowledgement transport protocol packet in the transport protocol packet buffer.

74. The messaging system as recited in claim 73, wherein the first program instructions are further executable within the first node to:

transmit a transport protocol request packet to the third node; and

wherein the third program instructions are further executable within the third node to:

receive the transport protocol request packet from the first node; and

transmit the acknowledgement transport protocol packet stored in the transport protocol packet buffer to the first node via the transport protocol tunnel connection in response to the received transport protocol request packet.

75. The messaging system as recited in claim 69, further comprising:

a third node comprising a third memory, wherein the transport protocol tunnel connection passes through the third node, wherein the third memory comprises third program instructions executable within the third node to:

receive the acknowledgement transport protocol packet transmitted to the first node via the transport protocol tunnel connection from the second node; and

forward the acknowledgement transport protocol packet to the first node.

76. The messaging system as recited in claim 59, wherein the first node is a server in the messaging system, and wherein the second node is a client in the messaging system.

77. The messaging system as recited in claim 59, further comprising:

a third node comprising:

a third memory; and

a transport protocol packet buffer;

wherein the transport protocol tunnel connection passes through the third node, wherein the third memory comprises third program instructions executable within the third node to:

receive the one or more transport protocol packets transmitted to the second node via the transport protocol tunnel connection from the first node; and

store the one or more transport protocol packets in the transport protocol packet buffer on the third node.

wherein the second program instructions are further executable within the second node to transmit one or more transport protocol request packets to the third node; and

wherein the third program instructions are further executable within the third node to:

receive the one or more transmitted transport protocol request packets; and

transmit the one or more transport protocol packets stored in the transport protocol packet buffer to the second node via the transport protocol tunnel connection in response to the received one or more transport protocol request packets.

78. The messaging system as recited in claim 59, wherein the transport protocol is Hypertext Transport Protocol (HTTP).

79. The messaging system as recited in claim 59, wherein the transport protocol is one of UDP (User Datagram Protocol), IrDA (Infrared Data Association), SNA (Systems Network Architecture), IPX (Internetwork Packet eXchange), and Bluetooth.

80. A carrier medium comprising program instructions, wherein the program instructions are computer-executable to implement:

establishing a transport protocol tunnel connection from a first node in a messaging system to a second node in the messaging system;

generating a messaging system message on the first node;

generating one or more transport protocol packets, wherein the one or more transport protocol packets each includes at least a part of the messaging system message; and

transmitting the one or more transport protocol packets to the second node via the transport protocol tunnel connection;

wherein the transport protocol tunnel connection provides full-duplex transmission of messaging system messages between the first node and the second node, and wherein the transport protocol tunnel connection further provides delivery of the messaging system messages in the sequence in which the messaging system messages are generated.

81. The carrier medium as recited in claim **80**, wherein the transport protocol tunnel connection passes through a Web server, wherein the second node is a broker in the messaging system, and wherein, in said transmitting the one or more transport protocol packets to the second node, the program instructions are further computer-executable to implement:

transmitting the one or more transport protocol packets to the Web server; and

the Web server forwarding the one or more transport protocol packets to the broker via a Transmission Control Protocol (TCP) connection portion of the transport protocol tunnel connection between the Web server and the broker.

82. The carrier medium as recited in claim **80**, wherein the first node is a client in the messaging system, wherein the second node is a broker in the messaging system, wherein the transport protocol tunnel connection passes through a proxy server and a Web server, and wherein, in said transmitting the one or more transport protocol packets to the broker via the transport protocol tunnel connection, the program instructions are further computer-executable to implement:

transmitting the one or more transport protocol packets from the client to the proxy server;

transmitting the one or more transport protocol packets from the proxy server to the Web server; and

the Web server forwarding the one or more transport protocol packets to the broker;

wherein the transport protocol tunnel connection passes through at least one firewall between the proxy server and the Web server.

83. The carrier medium as recited in claim **80**, wherein the acknowledgement transport protocol packet includes information indicating available space in the receive buffer, and wherein the program instructions are further computer-executable to implement:

receiving the transmitted one or more transport protocol packets on the second node;

storing the messaging system message from the one or more transport protocol packets in a receive buffer on the second node;

generating on the second node an acknowledgement transport protocol packet to indicate successful receipt of the one or more transport protocol packets including the messaging system message;

transmitting the acknowledgement transport protocol packet to the first node via the transport protocol tunnel connection;

receiving the transmitted acknowledgement transport protocol packet on the first node;

generating one or more messaging system messages on the first node;

storing the one or more messaging system messages in a transmit buffer on the first node;

determining from the information indicating available space in the receive buffer included in the received acknowledgement transport protocol packet that there is space available to receive the one or more messaging system messages on the second node;

generating a second one or more transport protocol packets, wherein the second one or more transport protocol packets include the one or more messaging system messages; and

transmitting the second one or more transport protocol packets to the second node via the transport protocol tunnel connection.

84. The carrier medium as recited in claim **80**, wherein the first node is a client in the messaging system, wherein the transport protocol tunnel connection passes through a Web server, and wherein the program instructions are further computer-executable to implement:

receiving the transmitted one or more transport protocol packets on the second node;

storing the messaging system message from the one or more transport protocol packets in a receive buffer on the second node;

the second node generating an acknowledgement transport protocol packet to indicate successful receipt of the one or more transport protocol packets including the messaging system message;

transmitting the acknowledgement transport protocol packet to the Web server;

storing the acknowledgement transport protocol packet in a transport protocol packet buffer on the Web server;

the client sending a transport protocol request packet to the Web server; and

the Web server transmitting the acknowledgement transport protocol packet stored in the transport protocol packet buffer to the client via the transport protocol tunnel connection in response to the transport protocol request packet.

85. The carrier medium as recited in claim **80**, wherein the first node is a broker in the messaging system, wherein the transport protocol tunnel connection passes through a Web server, and wherein the program instructions are further computer-executable to implement:

receiving the transmitted one or more transport protocol packets on the second node;

storing the messaging system message from the one or more transport protocol packets in a receive buffer on the second node;

the second node generating an acknowledgement transport protocol packet to indicate successful receipt of the one or more transport protocol packets including the messaging system message;

transmitting the acknowledgement transport protocol packet to the Web server; and

the Web server forwarding the acknowledgement transport protocol packet to the first node via a Transmission Control Protocol (TCP) connection portion of the transport protocol tunnel connection.

86. The carrier medium as recited in claim **80**, wherein the second node is a client in the messaging system, wherein the transport protocol tunnel connection passes through a Web server, and wherein, in said transmitting the one or more transport protocol packets to the second node via the transport protocol tunnel connection, the program instructions are further computer-executable to implement:

transmitting the one or more transport protocol packets to the Web server;

storing the one or more transport protocol packets in a transport protocol packet buffer on the Web server;

the client sending one or more transport protocol request packets to the Web server; and

the Web server transmitting the one or more transport protocol packets stored in the transport protocol packet buffer to the client via the transport protocol tunnel connection in response to the one or more transport protocol request packets.

87. The carrier medium as recited in claim **80**, wherein the transport protocol is Hypertext Transport Protocol (HTTP).

* * * * *

US 20020161904A1

(19) **United States**

(12) **Patent Application Publication** (10) **Pub. No.: US 2002/0161904 A1**

Tredoux et al. (43) **Pub. Date: Oct. 31, 2002**

(54) **EXTERNAL ACCESS TO PROTECTED DEVICE ON PRIVATE NETWORK**

(75) Inventors: **Gavan Tredoux**, Rochester, NY (US); **Xin Xu**, Palo Alto, CA (US); **Bruce C. Lyon**, Victor, NY (US); **Randy L. Cain**, Plano, TX (US)

Correspondence Address:
Patent Documentation Center
Xerox Corporation
Xerox Square 20th Floor
100 Clinton Ave. S.
Rochester, NY 14644 (US)

(73) Assignee: **Xerox Corporation**

(21) Appl. No.: **09/845,104**

(22) Filed: **Apr. 30, 2001**

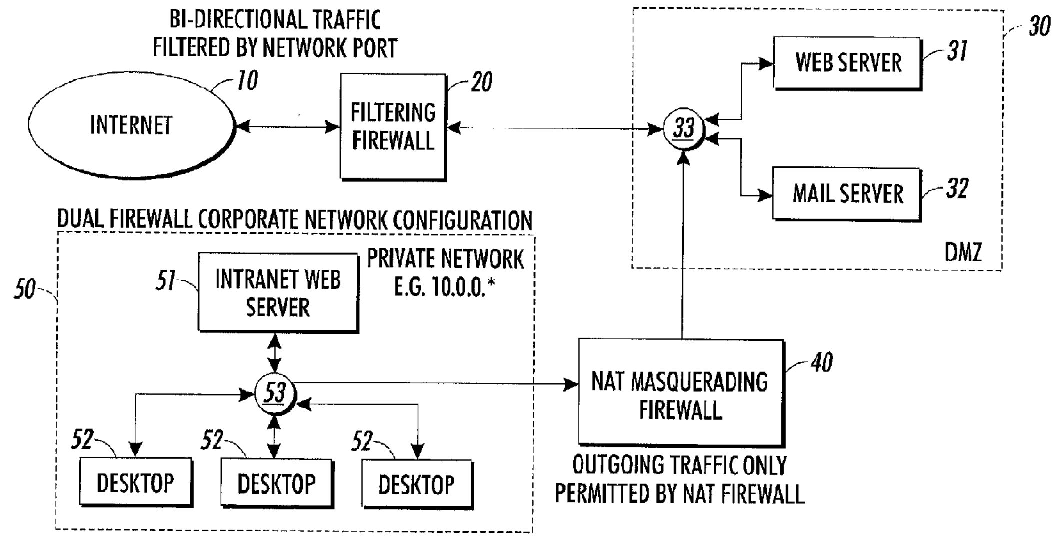
Publication Classification

(51) **Int. Cl.^{7}** **G06F 15/16**; G06F 12/14

(52) **U.S. Cl.** **709/229**; 713/200

(57) **ABSTRACT**

A scheme allowing communication between a network device on a protected network and an external network device outside the protected network using "reverse proxying." A proxy server receives incoming data on behalf of the protected network device, which data is retrieved by a proxy agent that periodically polls the proxy server to see if any data awaits retrieval.


BI-DIRECTIONAL TRAFFIC FILTERED BY NETWORK PORT
10
INTERNET
20
FILTERING FIREWALL
30
WEB SERVER
31
33
MAIL SERVER
32
DMZ
DUAL FIREWALL CORPORATE NETWORK CONFIGURATION
50
51
INTRANET WEB SERVER
PRIVATE NETWORK E.G. 10.0.0.*
53
40
NAT MASQUERADING FIREWALL
52
DESKTOP
52
DESKTOP
52
DESKTOP
OUTGOING TRAFFIC ONLY PERMITTED BY NAT FIREWALL

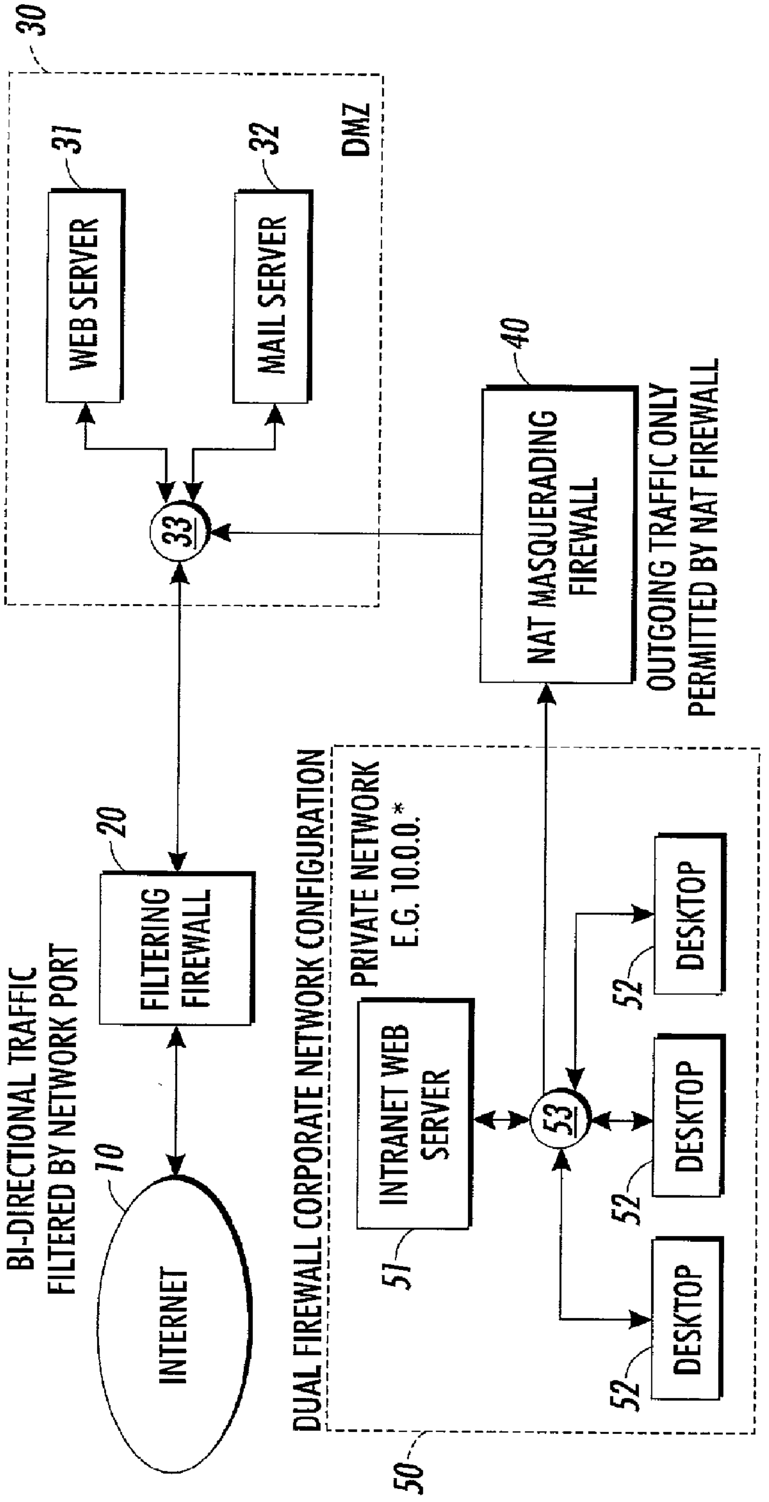

BI-DIRECTIONAL TRAFFIC
FILTERED BY NETWORK PORT
10
INTERNET
20
FILTERING FIREWALL
30
31
WEB SERVER
32
MAIL SERVER
DMZ
33
40
NAT MASQUERADING FIREWALL
OUTGOING TRAFFIC ONLY
PERMITTED BY NAT FIREWALL
DUAL FIREWALL CORPORATE NETWORK CONFIGURATION
PRIVATE NETWORK
E.G. 10.0.0.*
51
INTRANET WEB SERVER
53
52
DESKTOP
52
DESKTOP
52
DESKTOP
50

FIG. 1

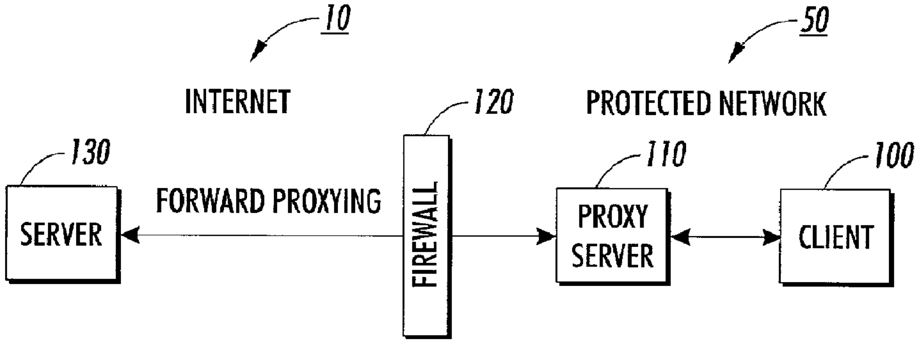

10
50
INTERNET
120
PROTECTED NETWORK
130
110
100
FORWARD PROXYING
SERVER
FIREWALL
PROXY SERVER
CLIENT

FIG. 2A

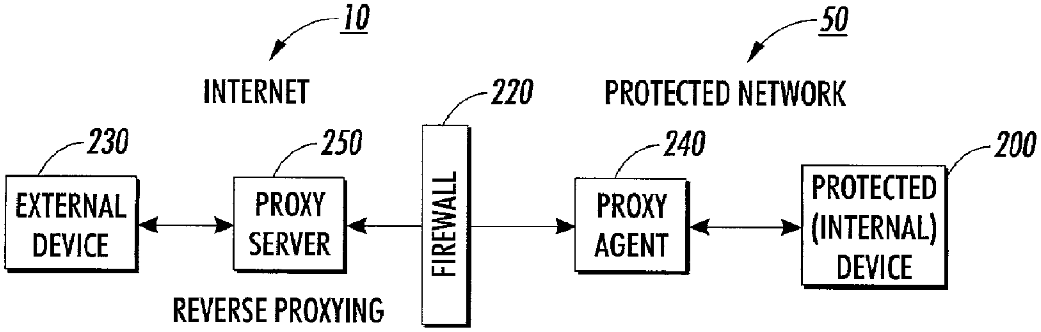

10
50
INTERNET
220
PROTECTED NETWORK
230
250
240
200
EXTERNAL DEVICE
PROXY SERVER
FIREWALL
PROXY AGENT
PROTECTED (INTERNAL) DEVICE
REVERSE PROXYING

FIG. 2B

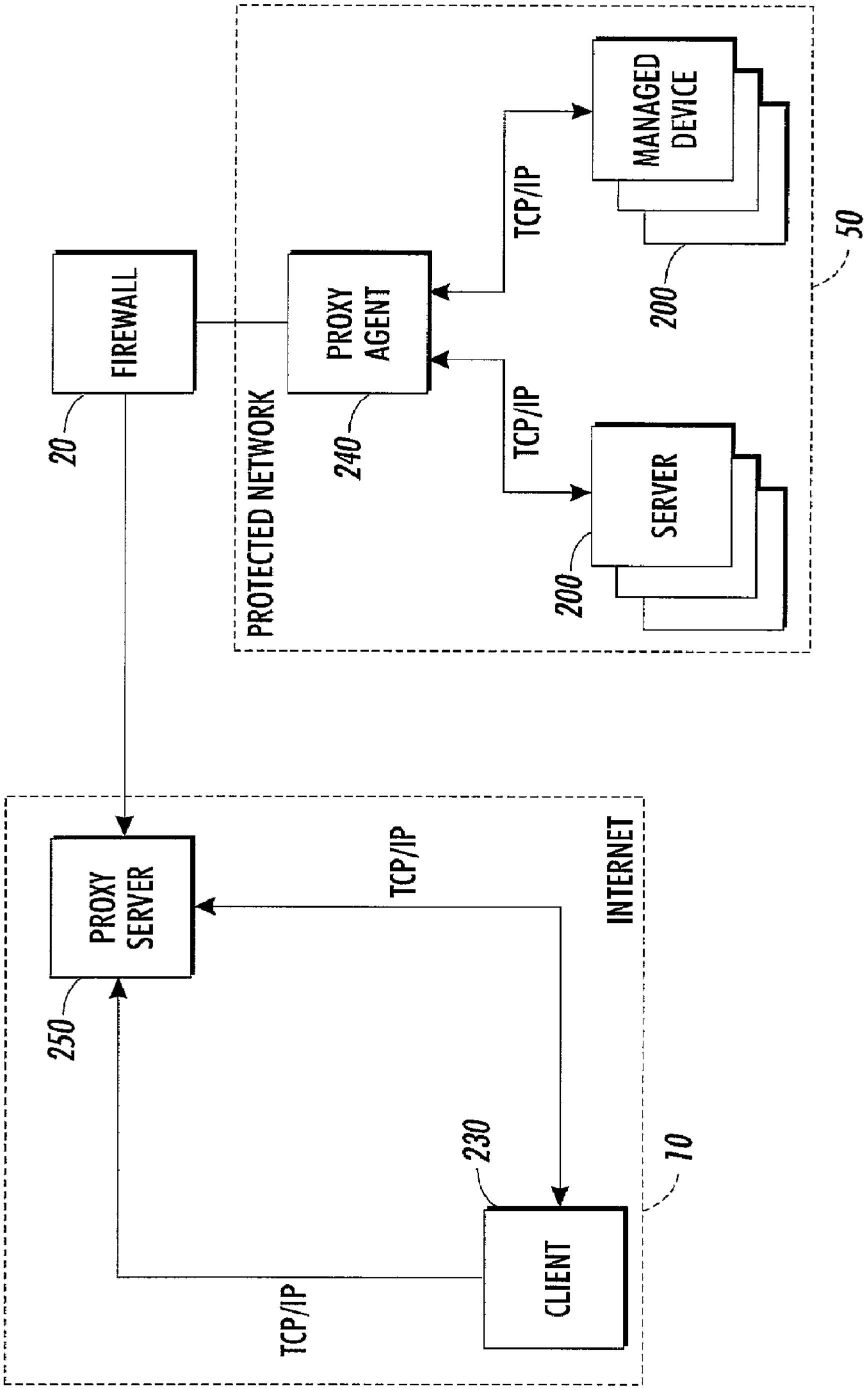
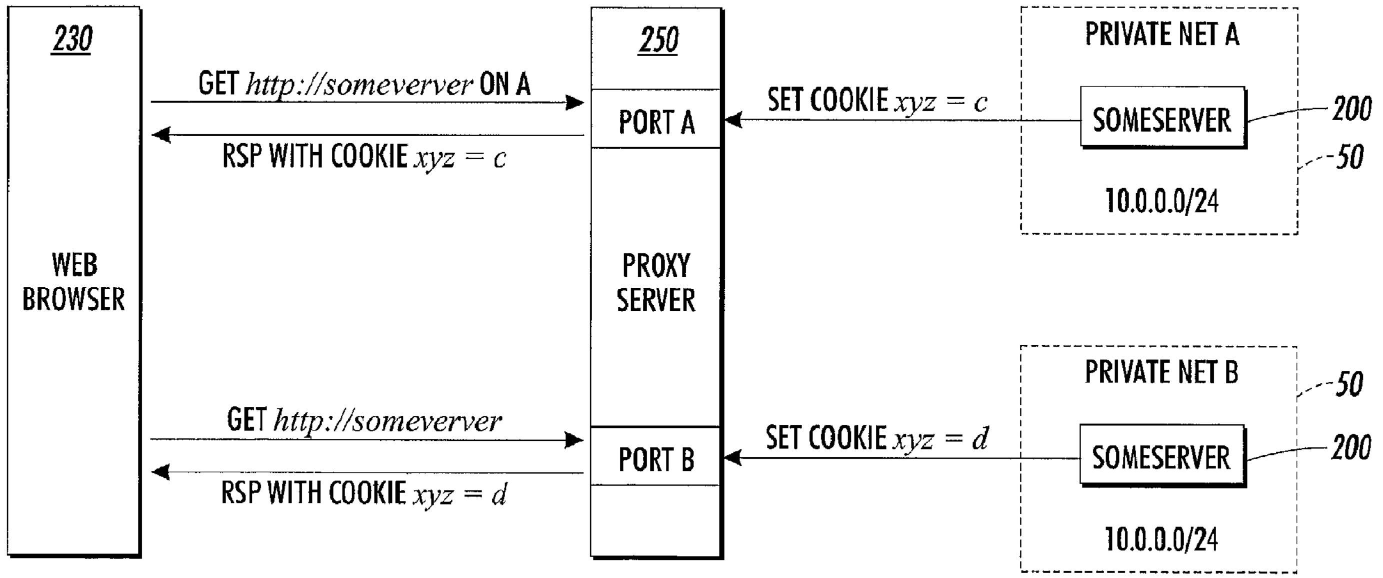

FIREWALL
20
PROXY AGENT
240
PROTECTED NETWORK
TCP/IP
TCP/IP
MANAGED DEVICE
200
SERVER
200
50
PROXY SERVER
250
TCP/IP
INTERNET
CLIENT
230
TCP/IP
10

FIG. 3

BROWSER IS CONFIGURED TO USE PROXY PORT A TO GET AT NETWORK A, THEN RECONFIGURED TO USE PORT B TO GET AT NETWORK B.

PROXY SERVER DETERMINES WHICH NETWORK A REQUEST IS SENT TO, E.G. USING A PORT BINDING SCHEME. THE BROWSER CAN BE CONFIGURED TO USE DIFFERENT PROXY PORTS.



BROWSER CANNOT DISTINGUISH URL *http://someserver* ON A FROM URL *http://someserver* ON B. HENCE COOKIE *xyz=d* FROM B OVERWRITES COOKIE *xyz=c* FROM A, SINCE IT IS RECEIVED LAST.

FIG. 4

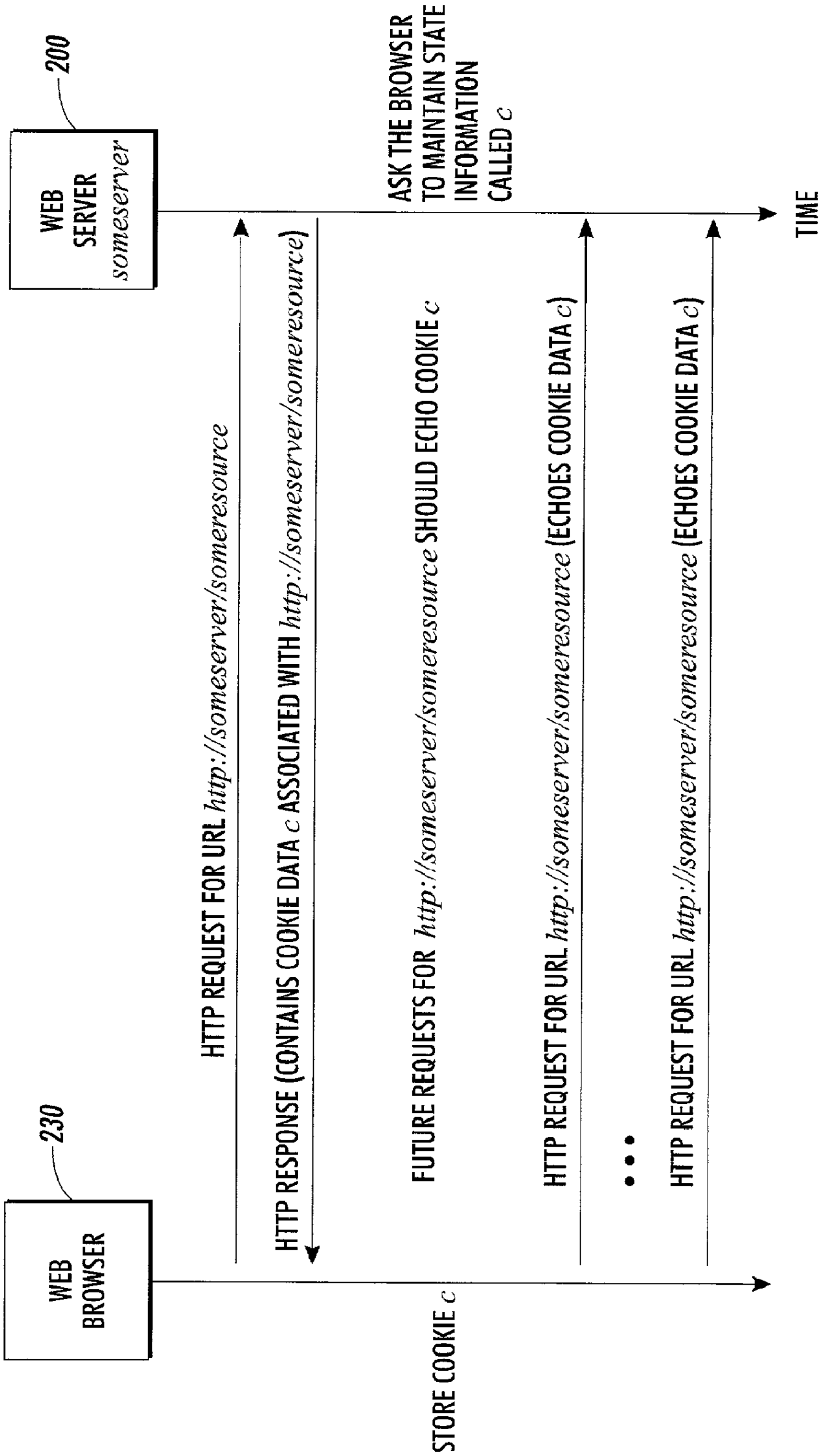

WEB BROWSER
230
WEB SERVER *someserver*
200
HTTP REQUEST FOR URL *http://someserver/someresource*
HTTP RESPONSE (CONTAINS COOKIE DATA *c* ASSOCIATED WITH *http://someserver/someresource*)
STORE COOKIE *c*
FUTURE REQUESTS FOR *http://someserver/someresource* SHOULD ECHO COOKIE *c*
ASK THE BROWSER TO MAINTAIN STATE INFORMATION CALLED *c*
HTTP REQUEST FOR URL *http://someserver/someresource* (ECHOES COOKIE DATA *c*)
•••
HTTP REQUEST FOR URL *http://someserver/someresource* (ECHOES COOKIE DATA *c*)
TIME

FIG. 5

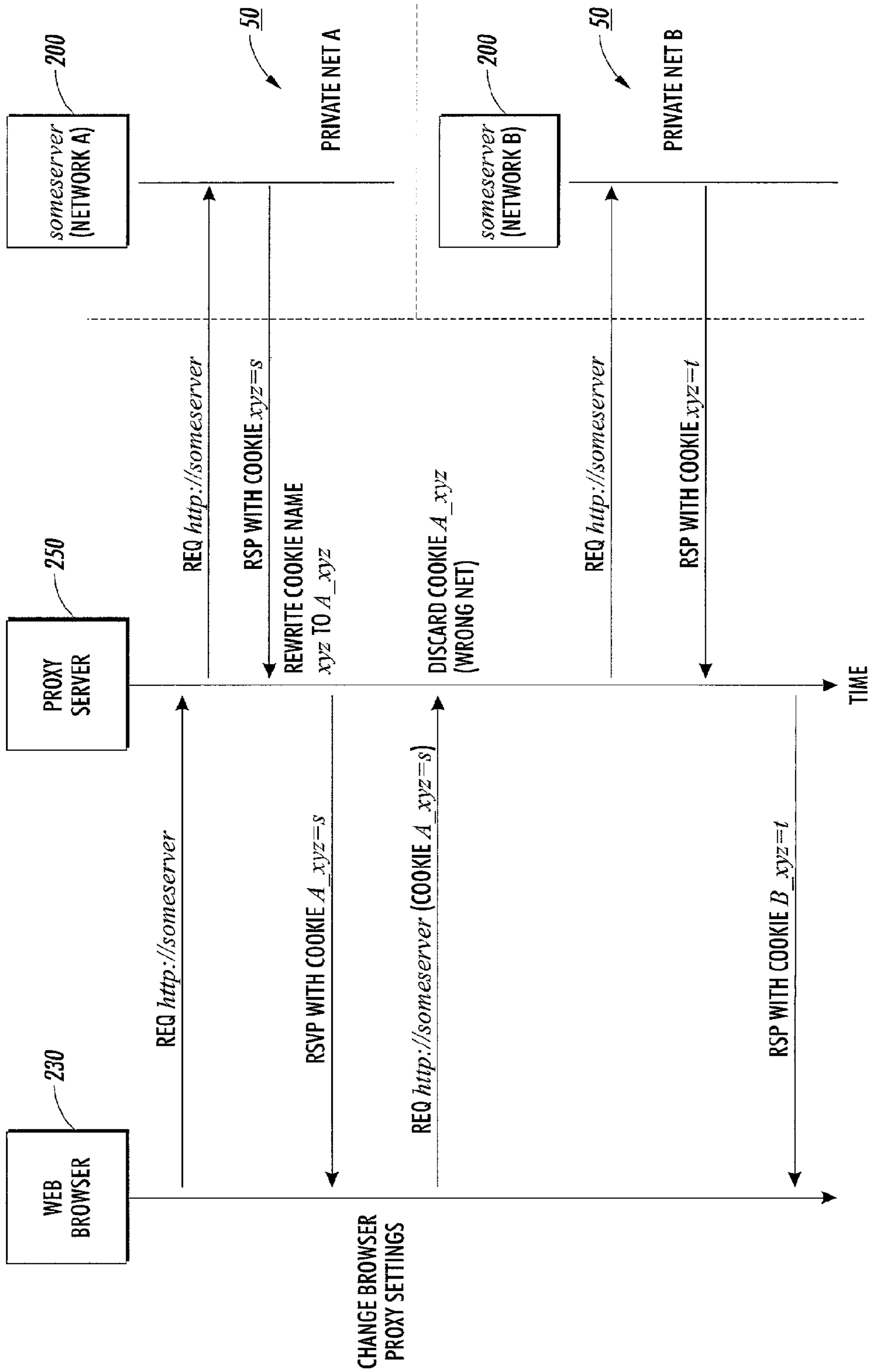

WEB BROWSER
230
PROXY SERVER
250
someserver (NETWORK A)
200
50
PRIVATE NET A
someserver (NETWORK B)
200
50
PRIVATE NET B
REQ http://someserver
RSVP WITH COOKIE A_xyz=s
REQ http://someserver (COOKIE A_xyz=s)
RSP WITH COOKIE B_xyz=t
CHANGE BROWSER PROXY SETTINGS
REQ http://someserver
RSP WITH COOKIE xyz=s
REWRITE COOKIE NAME xyz TO A_xyz
DISCARD COOKIE A_xyz (WRONG NET)
REQ http://someserver
RSP WITH COOKIE xyz=t
TIME

FIG. 6

EXTERNAL ACCESS TO PROTECTED DEVICE ON PRIVATE NETWORK

FIELD OF THE INVENTION

[0001] The present invention relates to protection and access protocols for networks such as computer networks and the like. In particular, the present invention relates to schemes allowing access to and from devices on protected networks from outside the protected networks.

BACKGROUND OF THE INVENTION

[0002] Networks connected to the Internet rely on firewalls and proxy servers to protect the networks against intrusion by unauthorized persons. Firewalls typically allow only incoming connections to designated machines and/or via particular protocols (TCP/IP, HTTP, FTP, etc.), disallowing all other traffic. Firewalls can also restrict traffic from the network to the Internet, as can outgoing proxy servers, by restricting destinations and/or protocols. However, these security restrictions often frustrate some uses of the Internet for legitimate purposes. For example, remote network equipment diagnosis and service is severely impaired, if not completely disabled, by firewalls.

[0003] Some firewalls can be modified and/or reconfigured to permit the traffic entry, but this can require the purchase of additional hardware and/or software. The cost associated with hardware and/or software purchase, combined with the difficulty of effecting a change in corporate policies regarding network security, would likely be a significant obstacle to the realization of such modifications. In addition, many firewalls and/or routers employ address masquerading and network address translation (NAT). Masquerading and NAT allow the use of internal network address spaces, but typically prevent incoming traffic from reaching the internal addresses since the internal addresses are non-routable and non-unique. No commercially-used or -available technique appears to solve all of these problems without modification of firewall/proxy server configurations, firewall/proxy server capabilities, and/or network security policies. For example, many virtual private network (VPN) schemes provide secure access between private networks via the Internet, but all require extensive modifications to the firewalls, proxy servers, and/or security policies of the connected networks.

SUMMARY OF THE INVENTION

[0004] Various embodiments of the invention allow traffic from outside a protected network to connect to an internal network device of the protected network through a firewall configured to protect the network. For example, TCP/IP traffic traveling to the protected network via the Internet can reach an intended computer on the internal network. The technique employed requires little or no alteration of the intended internal network device, firewall, proxy server, or security policy configurations, so long as outgoing connections are permitted via at least one protocol, such as, for example, HTTP. The outgoing connections can be made via a proxy server if necessary. Yet, even though the outgoing connection can be limited to one protocol, incoming traffic is not limited to the one protocol and can employ any protocol the Internet and the protected network, and the intended device, are capable of transmitting and/or handling. Public addressability of the protected network is not required, yielding access to the private, non-unique address space that is not ordinarily routable from clients outside the protected network. Still, the technique preserves network security via several built-in security measures.

[0005] The technique applied by various embodiments of the invention is referred to as "Reverse Proxying," in part because it includes two primary components: a proxy agent, located within the protected network; and an external proxy server, located outside the protected network (for example, on the Internet) at a location reachable by the proxy agent. The external proxy server stores traffic addressed to devices within the protected network until a proxy agent discovers queued traffic intended for the protected network, at which point the external proxy server forwards this traffic to the intended internal network device(s). In turn, the proxy agent forwards any responses it receives from the internal network device(s) back to the external proxy server, which transmits the responses to the intended clients.

[0006] The external proxy server represents clients connecting to the internal (protected) network devices; for example, clients can establish TCP/IP connections to the proxy server and send and receive data to the external proxy server on designated TCP/IP ports that are, in effect, forwarded by the external proxy server to the proxy agent. Likewise, the proxy agent connects to the otherwise inaccessible internal network devices, and sends/transmits and receives data as if it were the client. To a real external client, the external proxy server is the internal network device—the external proxy server thus masquerades as, or "pretends to be," the internal network device. To an internal network device, the proxy agent is the external client—the proxy agent thus masquerades as, or "pretends to be," the client. The link between the external proxy server and the proxy agent is transparent to both the external client and to the internal network device, and is of no concern to them.

[0007] To effect the transparent connection, various embodiments of the invention employ "trickle down polling" to reduce latency and provide highly responsive service without imposing the high network loads that can result from too-frequent polling. In addition, several security measures can be built-in to ensure that it cannot be used to compromise the integrity and privacy of the networks it services, up to the highest standards met by current Internet applications. For example, communication between the proxy agent and the external proxy server can be encrypted using an encryption system, such as the industry standard Secure Sockets Layer (SSL) for HTTP, preventing eavesdropping. Authentication of both the agent and the Server can be enforced by requiring, for example, X.509 certificates of both, or using another authentication technique, such as other "public key" based cryptography systems, and can be verified by a trusted certification authority. The external proxy server also implements a cookie rewriting process, ensuring that all cookies have truly unique identifiers; if a browser should attempt to transmit a cookie to a destination for which it is not intended, the external proxy server will silently drop the cookie from the request. Further, network administrators can be given fine-grained control over the Reverse Proxying system.

[0008] More specifically the present invention relates to a reverse proxy network communication scheme wherein a proxy agent located inside a protected network is addressable by internal network devices. The proxy agent establishes outgoing network connections on behalf of the internal network devices through a security device, such as a firewall, through which all traffic between the protected network and external networks, such as networks and external network devices on the Internet, must travel. The security device permits at least outgoing connections via at least one predetermined network protocol, such as HTTP.

[0009] An external proxy server outside the protected network is reachable by the proxy agent via outgoing network connections through the security device. The external proxy server is addressable by external network devices, thereby allowing communication between the external network devices and the internal network devices.

BRIEF DESCRIPTION OF THE DRAWINGS

[0010] This disclosure includes the attached Figures, which Figures are summarized as follows:

[0011] **FIG. 1** illustrates a typical protected network connected to the Internet.

[0012] **FIG. 2A** shows a simplified schematic of the connections between a client machine on a protected network and a sever on the Internet.

[0013] **FIG. 2B** shows a simplified schematic of the connections between a client machine on the Internet and a server on a protected network according to principles of the invention described in this application.

[0014] **FIG. 3** shows a more detailed schematic of the connections between client machines and servers on protected networks according to principles of the invention described in this application.

[0015] **FIG. 4** depicts two exemplary private networks, to which a web browser is connected, through a reverse proxy server. The two distinct networks have identical private network addresses, and the figure shows how cookies originating from these networks may be confused by the browser.

[0016] **FIG. 5** shows an exemplary timeline of an HTTP cookie protocol that can be used in embodiments of the invention where a browser connects to a unique network address space.

[0017] **FIG. 6** shows an exemplary timeline of an HTTP cookie protocol that can be used in embodiments of the invention where cookies from duplicate private network address spaces are confused.

DETAILED DESCRIPTION OF THE INVENTION

[0018] In various embodiments of the invention, communication between a device internal to a protected network and a device external to a protected network can be achieved where conventional security devices, such as firewalls and/or proxy servers, would not allow such communication. For example, incoming TCP/IP connections from a network **10**, such as the Internet, outside a firewall-protected network **50** to protected/internal devices on the protected network can occur. The technique used in various embodiments requires no alteration of the firewall **20** configuration or existing security policies, provided that the firewall **20** permits outgoing HTTP connections from the protected/internal device. Incoming connections are not restricted to any particular protocol, such as HTTP, but may be any appropriate networking protocol, including, but not limited to, FTP, gopher, smtp, pop, http, rtsp, and IPX. The outgoing connections are not limited to HTTP, but can be any appropriate protocol the networks, firewall, and/or proxy servers can handle. No alteration of the devices typically connected to a protected network is required, nor does a system deployed according to the principles of the invention require that the protected network **50** be publicly addressable. The technique employed will function unaltered in a private, non-unique address space not ordinarily routable for clients on the Internet **10**. Several built-in security measures maintain the privacy of the firewalled network.

[0019] **FIG. 1** illustrates a highly secure network configuration with dual firewalls **20**, a public "Demilitarized Zone" (DMZ) segment, and a private address space completely inaccessible to outside hosts. Devices and servers for internal use would be hosted on the private segment and would therefore ordinarily be totally isolated from the Internet **10**.

[0020] Applying the techniques of various embodiments of the invention, network traffic between external network devices and internal network devices hidden behind the security device **20** is possible even though the protected network uses a private address space. For example, embodiments similar to that shown in **FIG. 2B** can have TCP/IP network connectivity between an external device and devices hidden behind firewalls **20**. The only assumption made is that outgoing connections, such as HTTP connections, are permitted by the existing firewall configurations, possibly through an outgoing proxy server, and by corporate security policies. No alterations are required to:

[0021] 1. The networked devices.

[0022] 2. The firewalls used to protect the network.

[0023] 3. Corporate security policies.

[0024] 4. The address spaces

[0025] 5. The clients used to connect to the hidden devices

[0026] 6. The TCP/IP protocol used by the client and server

[0027] The absence of such alterations can render the processes of the present invention easy and inexpensive to deploy, with substantially no disruption of the existing network, which can be a considerable improvement over existing solutions.

[0028] As illustrated in, for example, FIGS. 2B, and **3-6**, "Reverse Proxying" primarily comprises two components: the proxy agent **240** and the external proxy server **250**. The proxy agent **240** is located within the protected network **50**. It is assumed that this agent has the ability to establish outgoing network connections, such as HTTP connections, possibly through an outgoing HTTP proxy server, to the Internet **10**. For the purposes of explaining the operation of embodiments of the invention, particular protocols will be used, but the invention is not limited to the particular protocols used in this example. The external proxy server **250** is located outside the protected network **50**, on the Internet **10**, at a location reachable by the agent and receives traffic addressed to internal network devices. The proxy agent **240** periodically polls the external proxy server **250** to check for queued traffic intended for the protected network **50**. When the proxy agent **240** discovers traffic intended for internal network devices, it forwards this traffic to the intended recipients. In turn, the proxy agent **240** will forward any responses it receives back to the external proxy server **250**, which will transmit the responses to the intended external network device clients. **FIG. 3** illustrates an embodiment of this architecture:

> **[0029]** For clients connecting to the hidden (protected) internal network devices, the external proxy server **250** represents those devices and thus masquerades as the internal network devices. In various embodiments of the invention, clients establish TCP/IP connections to the proxy server **250**, and send and receive data to the external proxy server **250**, on designated TCP/IP ports that are, in effect, forwarded by the external proxy server **250** to the proxy agent **240**. Likewise, the proxy agent **240** connects to the otherwise hidden internal network devices, and sends and receives data as if it is the external network device client. Thus, the proxy agent **240** masquerades as the external network device client. The link between the external proxy server **250** and the proxy agent is transparent to both the external network device client and the internal network device, and is of no concern to them.

[0030] As mentioned above, in various embodiments of the invention, connections and data received by the external proxy server **250** are stored for later retrieval by the proxy agent **240**. The proxy agent polls the external proxy server **250** at regular intervals, using, for example, an HTTP connection, to discover pending connections and data, and deliver responses from the intended internal network devices. In effect, the TCP/IP traffic between the external network device client and the internal network device is "tunneled" through HTTP in this way, encapsulated in HTTP requests and responses with header information indicating the source and destination IP addresses and the intended ports. To improve efficiency, multiple requests can be multiplexed through the same HTTP connection.

[0031] It is instructive to compare the Reverse Proxying, with traditional "forward" proxying. **FIGS. 2A and 2B** illustrate the difference between traditional proxying **(FIG. 2A)** and the reverse proxying employed by embodiments of the invention **(FIG. 2B)**.

[0032] Providing access to private IP addresses is what allows the success and generality of this scheme. The private IP address spaces **50** are not unique across the Internet **10** and many different organizations reuse the same IP address spaces **50**. For the IP address spaces **50** and the internal network devices **200** residing therein to be addressable by external network device clients **230**, the external proxy server **250** maintains a map between local TCP/IP ports on the proxy server **250** and remote private IP addresses distinguished by the identify of the proxy agent used to access them. Proxy agents publish a list of addresses they can reach to the external proxy server **250**, and this list is used by the external proxy server **250** to establish the map between local ports and agents/remote addresses.

[0033] No assumptions need be rendered regarding the network protocol used by the external network device client to communicate with the internal network device and/or (hidden) server on the protected network. All network traffic, for example TCP/IP traffic, is tunneled by the proxy agent **240** through the exemplary HTTP connection between the proxy agent **240** and the external proxy server **250**, and there is generally no need for them to alter this data, with some notable exceptions. Certain protocols can require special treatment, particularly HTTP itself. The use of embedded hyperlinks in HTML pages implies that a client may be redirected by a link to an inaccessible URL hidden behind the security device/firewall **20**, away from the external proxy server **250** which enables its access to the hidden network. To prevent or minimize such undesirable redirection, a web browser/external client device **230** can be configured (through standard browser settings) to use the external proxy server **250** as a true HTTP proxy server, using the local port on the server described above. This ensures that all HTTP requests are forwarded intact and uninterpreted to the external proxy server **250**, which passes those requests to the proxy agent **240**. The agent **240** retrieves the requested URLs, which are directly accessible to it since it is behind the firewall **20**.

[0034] The proxy agent **240** is forced to poll the external proxy server **250** for pending traffic because it is assumed that only outgoing HTTP connections are permitted by the network security device **20**. This introduces a latency problem, since the polling interval determines the responsiveness of the TCP/IP traffic tunneled over the polled HTTP connection. Latency refers to delays introduced by the time it takes for traffic to travel from an origin to a destination and from the destination back to the origin. Since traffic must be queued by the proxy server until the proxy agent polls it, there is a delay between arrival of the traffic at the proxy server and arrival at the proxy agent, increasing the latency. High latency, delays on the order of tenths of a second or more, between requests and responses can compromise the practical usability of a system employing reverse proxying. Latency can be reduced by a decreased polling interval, but this imposes an increasing network load burden and can be limited by the minimum time required to establish and complete an outgoing HTTP request.

[0035] To reach a suitable compromise between latency reduction and network load, various embodiments of the invention employ "trickle down polling to reduce latency and provide highly responsive service without imposing the high network loads implied by too-frequent polling. The proxy agent **240** connects to the external proxy server **250** to discover pending traffic. If there is nothing pending, the external proxy server **250** returns a slow stream of spurious bytes which are ignored by the proxy agent **240**. When the external proxy server **250** receives data from an external network device or client/browser **230**, it is immediately transmitted to the proxy agent **240** and the connection is closed to flush any buffering performed by intervening (outgoing) proxy servers. To improve response times, the agent **240** can open several connections to the proxy server **250** to reduce the likelihood that no connections will be open when traffic arrives. The trickling-down of spurious bytes prevents any timeouts on the outgoing HTTP request, which may be enforced by intervening outgoing proxy servers. In this way, highly responsive service is guaranteed since the proxy agent **240** can usually be informed immediately of incoming traffic, removing the undesirable latency between the time that this traffic is queued on the external proxy server **250** and the time that the proxy agent **240** retrieves it. However, the Internet **10** itself can impose a lower bound on latency since it can determine the time taken to transmit requests and responses, and network protocols used by the Internet, such as TCP/IP, do not provide guaranteed service.

[**0036**] Several security measures can be built into the invention to ensure that it cannot be used to compromise the integrity and privacy of the networks it services, up to the highest standards met by current Internet applications.

[**0037**] Communication between the proxy agent **240** and the external proxy server **250** can, for example, be encrypted using an encryption system, such as the industry standard Secure Sockets Layer (SSL) for HTTP, preventing eavesdropping. Authentication of both the agent **240** and the server **250** can be enforced by requiring, for example, X.509 certificates of both, or using another authentication technique, such as other "public key" based cryptography systems, and can be verified by a trusted certification authority. The external proxy server **250** can also implement a cookie rewriting process, such as the exemplary process illustrated in FIGS. **4**-**6**, ensuring that all cookies have truly unique identifiers.

[**0038**] As shown in **FIG. 5**, web servers **200** can request that clients **230** (web browsers) maintain state through a mechanism known as "cookies". To effect cookies, servers insert additional headers onto replies to HTTP requests, which specify named "echo" data that the browser should repeat back to the server when accessing certain resources identified in the header. Each data element to be stored and echoed is called a "cookie."

[**0039**] Following such a cookie protocol, a web browser associates cookies with the Uniform Resource Locators (URLs) to which they were bound by the web server. In normal Internet usage, these URLs are guaranteed to be unique. However, in a reverse proxying situation, in which private network addressing becomes a factor, these URLs are not necessarily unique—this is true whether or not IP addresses or symbolic names are used in the URL, since symbolic domain names need not be unique across private IP spaces. This can create two problems:

[**0040**] 1. Race conditions. In this situation, the browser overwrites an existing cookie for a URL with the most recent value tied to that URL. There is consequently a race between servers to set the cookie data. A server that associates cookie data with a URL is thus not guaranteed that it will receive the same data back. This can partially or totally disable web servers/applications that rely on correct state data echoed in cookies.

[**0041**] 2. Privacy violations. In this situation, cookie data associated with a URL can contain private data from a protected network, since servers in such networks can assume that all transmission between themselves and clients is secured. However, the browser could now unwittingly transmit this private data to a wholly different network, since it confuses the non-unique URLs. Servers in the wrong network might therefore gather sensitive data from other private networks, intentionally or unintentionally, in this way. This can be a serious compromise of the network security established by the firewall/private IP space system.

[**0042**] **FIG. 4** illustrates how cookies from different networks can be confused by web browsers. Web clients (browsers) **230** use URLs to uniquely identify resources on the Internet **10**. This is both specified by the relevant standards and by common practice. However, by providing access to private/protected networks **50** with not-necessarily-unique URLs,. reverse proxying schemes create potential confusion between these URLs. This only becomes an issue, however, when a stored state is associated with a (non-unique) URL(s) and transmitted later as part of requests for other networks, since all current requests are explicitly directed to the proper destinations by the proxy server configuration. This situation is analogous to luggage-handling errors on airline flights, where the incorrect luggage is transported on a flight that is directed to an otherwise-correct destination, due to a non-unique label on the luggage.

[**0043**] In various embodiments of the invention, a process referred to as "cookie rewriting" eliminates cookie ambiguity. All cookies have names. Typically, proxy servers do not alter any data sent or received by proxy. In various embodiments, the invention makes an exception for cookie names, which are rewritten by the proxy server as they are transmitted back to browsers for storage, to indicate clearly which private network they originate from. The reverse proxying scheme has some way of distinguishing private networks in embodiments of the invention (e.g. by the identity of the agent within those networks which effects firewall traversal) or the proxy server would not function correctly. One way of doing this is to prepend the unique identity of the private network to each cookie name (that is, place the private network identifier at the "front" of the cookie as a "prefix"), which is the implementation used in various embodiments of the invention, though other rewriting methods are possible. The prefix can then be stripped from the cookie when it is transmitted. Cookies passed by the browser with a request which originated from a different network are silently dropped by the proxy server. Thus the external proxy server maintains the privacy of the networks and ensures correct cookie storage and passing by browsers.

[**0044**] In the situation shown in **FIG. 6, a** browser first issues an HTTP GET request for the URL http://someserver, via the Proxy Server. The browser is configured to use Port A on the Proxy Server, which associates Port A with the private network A. The Proxy Server performs the request on the behalf of the browser (using whatever firewall traversal scheme it supports), and inspects any cookies which the someserver returns in the response. In this case, the cookie xyz with the value s has been set by someserver. The Proxy server rewrites the name of the cookie to A_xyz so it is clearly marked as a cookie intended for private network A. Note that the web browser attaches no intrinsic meaning to cookie names, simply echoing them to the URLs they are associated with. The browser receives the HTTP response from the proxy server, and stores the cookie A_xyz=s.

[**0045**] Later the browser is reconfigured to use Port B on the Proxy Server, which associates port B with the private network B. The browser issues an HTTP GET request for the same URL http://someserver, sending the cookie A_xyz=s with the request. It does so because it has no way of determining that the intended network has changed. The Proxy Server inspects any cookies contained in the request before forwarding it to someserver in the network B. Since the cookie A_xyz=s is intended for A and not B, it is discarded by the Proxy Server, and the rest of the request is forwarded. As before, the Proxy Server rewrites the names of any cookies contained in the HTTP response, so that xyz=t becomes B_xyz=t. This ensures that, in future, the cookie will not be passed to the network A, or any other network it was not intended for.

[0046] In addition to the above security measures, network administrators can be given fine-grained control over the Reverse Proxying system. For example, administrators can be granted the authority and/or ability to allow or deny entry into their network on a per-session basis by granting a permission, such as a short-lived key; administrators can also be granted the authority and/or ability to completely disable access, or limit it by other criteria.

[0047] The preceding description of the invention is exemplary in nature as it pertains to particular embodiments disclosed and no limitation as to the scope of the claims is intended by the particular choices of embodiments disclosed.

[0048] Other modifications of the present invention may occur to those skilled in the art subsequent to a review of the present application, and these modifications, including equivalents thereof, are intended to be included within the scope of the present invention.

What is claimed is:

1. A reverse proxy network communication scheme comprising:

a proxy agent located inside a protected network addressable by a least one internal network device, the proxy agent establishing outgoing network connections;

a security device through which all traffic between the protected network and external networks must travel, the security device permitting at least outgoing connections via at least one predetermined network protocol;

an external proxy server outside the protected network and reachable by the proxy agent via outgoing network connections through the security device, the external proxy server also being addressable by at least one external network device, thereby allowing communication between the at least one external network device and the at least one internal network device.

2. The scheme of claim 1 wherein the at least one predetermined network protocol is HTTP.

3. The scheme of claim 1 further including an outgoing proxy server in communication with the proxy agent and which the proxy agent uses to establish outgoing connections.

4. The scheme of claim 1 wherein the external proxy server is in communication with at least one other network, receives, and stores data addressed to the at least one internal network device.

5. The scheme of claim 4 wherein the proxy agent polls the external proxy server to check for data addressed to the at least one internal network device.

6. The scheme of claim 5 wherein the proxy agent downloads data addressed to the at least one internal network device from the external proxy server and forwards the data to the at least one internal network device.

7. The scheme of claim 4 wherein the external proxy server ensures proper cookie routing.

8. The scheme of claim 1 wherein the proxy agent forwards outgoing data to the external proxy server, which transmits the data to the at least one external network device.

9. A method of accessing an internal network device on a protected network, the network including a security device, the method comprising:

storing data addressed to the internal network device in an external proxy server;

maintaining a proxy agent on the protected network, the proxy agent executing the step of:

polling the external proxy server for data addressed to the internal network device;

forwarding to the internal network device any data on the external proxy server and addressed to the internal network device; and

forwarding to the external proxy server any data addressed to an external device in communication with the external proxy server.

10. The method of claim 9 further comprising polling the external proxy server at regular intervals.

11. The method of claim 9 further comprising communicating by the internal network device with the external proxy server using a first network protocol and the external network device communicates with the external proxy server using a second network protocol.

12. The method of claim 11 wherein data addressed to the internal network device using the second network protocol is transmitted to the internal device using the first network protocol so that the second network protocol is carried to the internal network device inside the first network protocol.

13. The method of claim 9 further including multiplexing multiple requests from the proxy agent to the external proxy server through the same connection.

14. The method of claim 9 further including maintaining by the external proxy server of maps between local TCP/IP ports of the external proxy server and private IP addresses on the protected network, the maps being distinguished by an identity of the proxy agent used to access them.

15. The method of claim 14 further including publishing by each proxy agent a list of addresses it can reach to the external proxy server, the external proxy server using this list to create a respective map between local ports and proxy agents.

16. The method of claim 14 further including ensuring cookie delivery.

17. The method of claim 9 wherein polling comprises:

connecting to the external proxy server to check for pending traffic;

returning a slow stream of spurious bytes ignored by the proxy agent if there is nothing pending;

immediately transmitting data from the external proxy server to the proxy agent when the external proxy server receives data from a client, thus closing the connection to flush any buffering performed by intervening (outgoing) proxy servers.

18. The method of claim 9 wherein communication between the proxy agent and external proxy server is encrypted.

19. The method of claim 18 wherein the data is encrypted using Secure Sockets Layer (SSL) for HTTP.

20. The method of claim 19 wherein both the proxy agent and the external proxy server require X.509 certificates.

21. The method of claim 9 further comprising rewriting cookies with unique identifiers to prevent inadvertent transmission of private information to an incorrect recipient on the protected network.

22. The method of claim 9 further comprising providing network administrators control over the system including granting administrators the ability to allow and deny entry into the protected network on a per session basis.

23. The method of claim 22 wherein access is conferred by granting a key with a predetermined life span.

* * * * *

US 20030182431A1

(19) **United States**

(12) **Patent Application Publication** (10) **Pub. No.: US 2003/0182431 A1**

Sturniolo et al. (43) **Pub. Date: Sep. 25, 2003**

(54) **METHOD AND APPARATUS FOR PROVIDING SECURE CONNECTIVITY IN MOBILE AND OTHER INTERMITTENT COMPUTING ENVIRONMENTS**

(76) Inventors: **Emil Sturniolo**, Medina, OH (US); **Aaron Stavens**, Auburn, WA (US); **Joseph T. Savarese**, Edmonds, WA (US)

Correspondence Address:
Nixon & Vanderhye P.C.
8th Floor
1100 North Glebe Road
Arlington, VA 22201 (US)

(21) Appl. No.: **10/340,833**

(22) Filed: **Jan. 13, 2003**

Related U.S. Application Data

(63) Continuation of application No. 09/330,310, filed on Jun. 11, 1999, now Pat. No. 6,546,425.

(60) Provisional application No. 60/347,243, filed on Jan. 14, 2002.

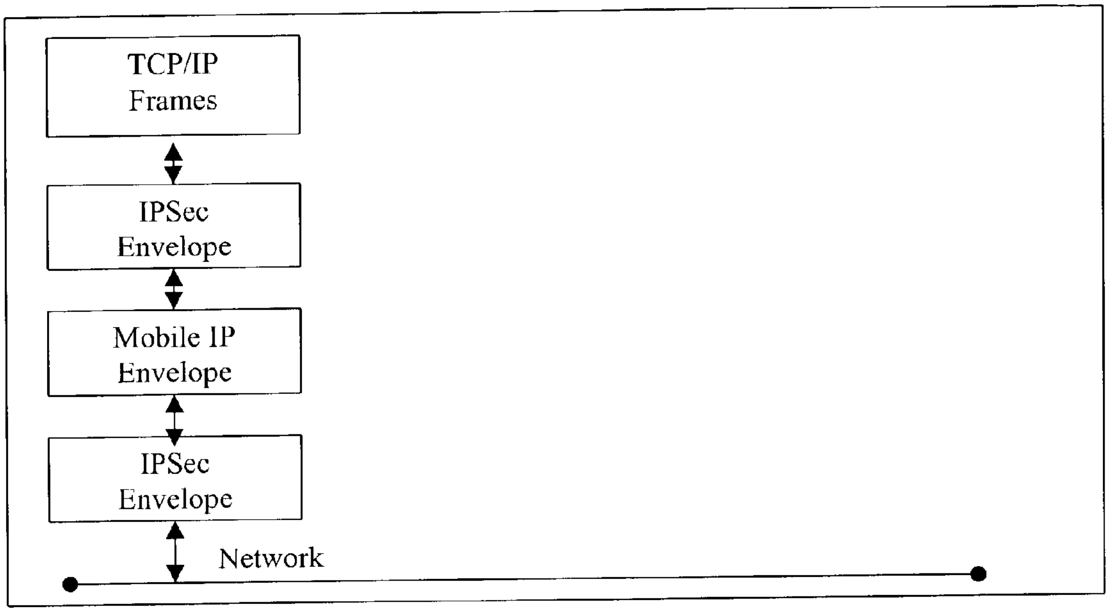
Publication Classification

(51) **Int. Cl.**^{7} **G06F 15/16**

(52) **U.S. Cl.** **709/227**; 713/200

(57) **ABSTRACT**

Method and apparatus for enabling secure connectivity using standards-based Virtual Private Network (VPN) IPSEC algorithms in a mobile and intermittently connected computing environment enhance the current standards based algorithms by allowing migratory devices to automatically (re)establish security sessions as the mobile end system roams across homogeneous or heterogeneous networks while maintaining network application session. The transitions between and among networks occur seamlessly—shielding networked applications from interruptions in connectivity. The applications and/or users need not be aware of these transitions, although intervention is possible. The method does not require modification to existing network infrastructure and/or modification to networked applications.



Enabling Roaming With Mobile IP And IPSec Encapsulation

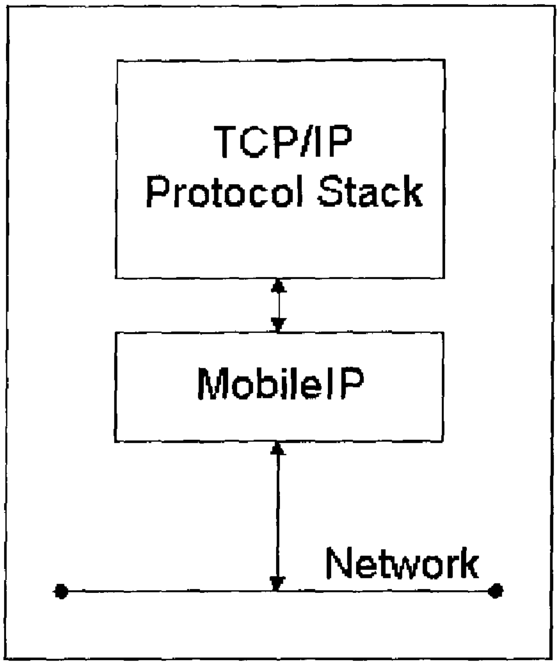


Figure 1 – Example Mobile IP Client Architecture (Prior Art)

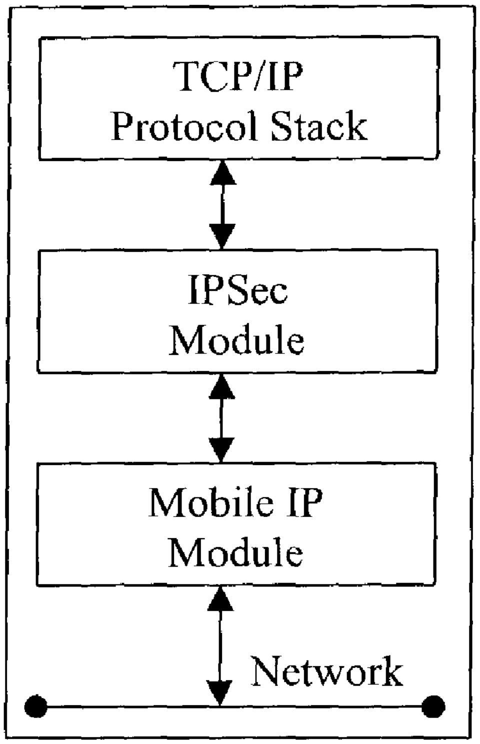


Figure 2: Example IPSec and Mobile IP Architecture (Prior Art)

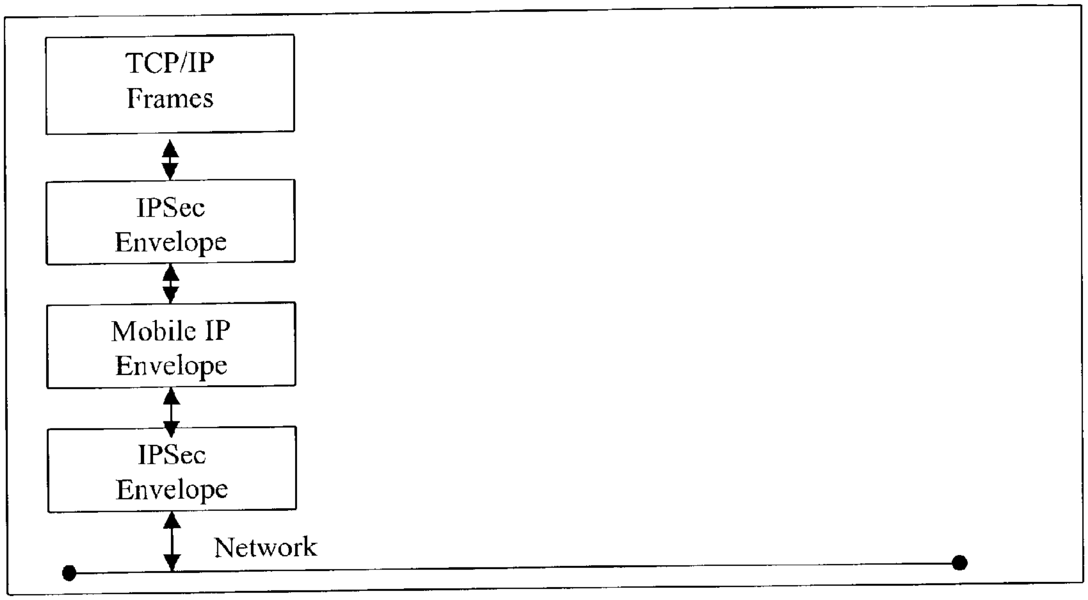


Figure 3: Enabling Roaming With Mobile IP And IPSec Encapsulation

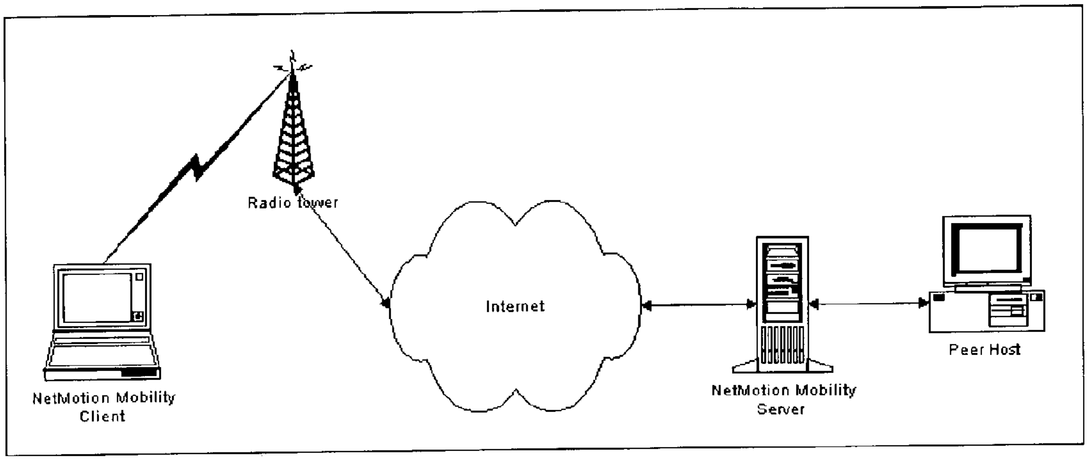


Figure 4 – Example Non-Limiting Secure Mobility Architecture

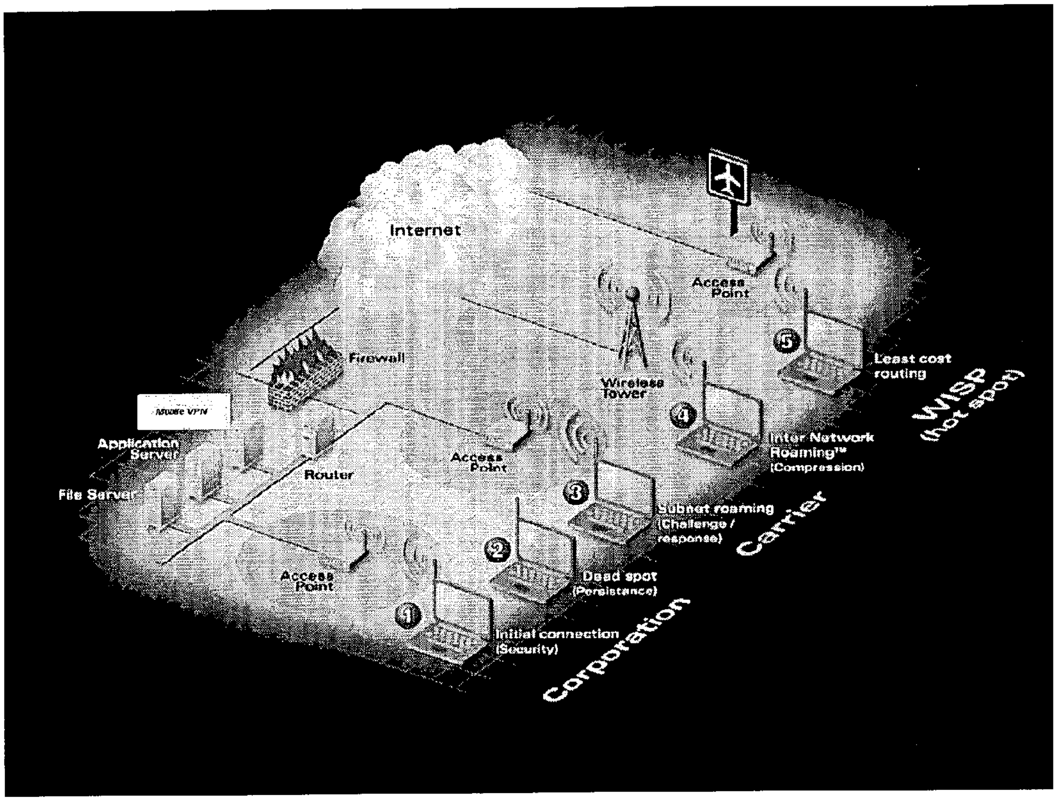


Figure 5 – Example Non-Limiting Illustrative Mobile Usage Scenarios

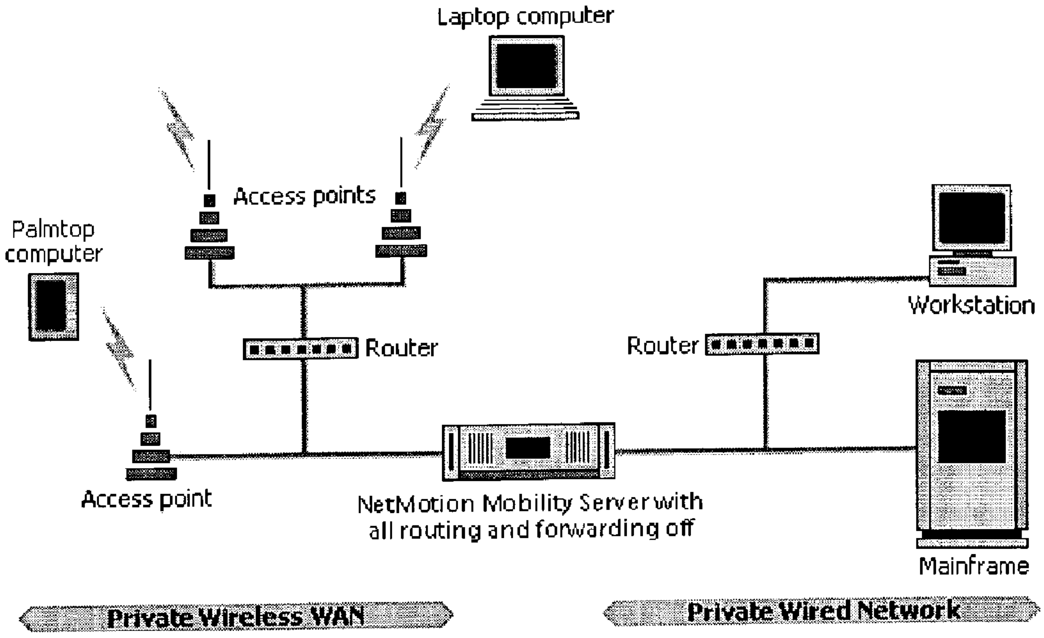

Laptop computer
Access points
Palmtop computer
Router
Router
Workstation
Access point
NetMotion Mobility Server with all routing and forwarding off
Mainframe
Private Wireless WAN
Private Wired Network

Figure 5A

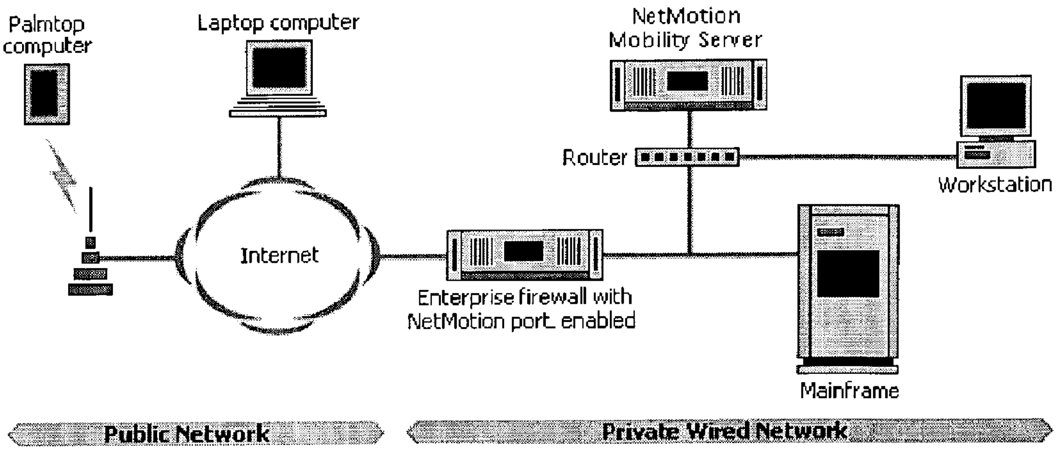

Palmtop computer
Laptop computer
NetMotion Mobility Server
Router
Workstation
Internet
Enterprise firewall with NetMotion port enabled
Mainframe
Public Network
Private Wired Network

Figure 5B

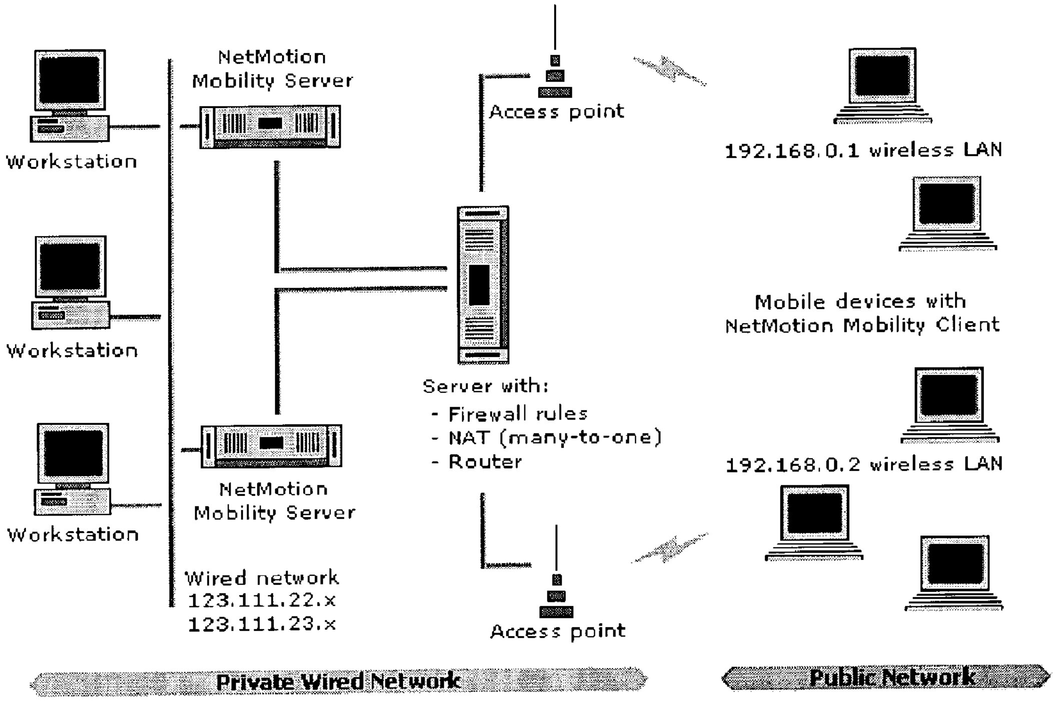

NetMotion
Mobility Server
Access point
Workstation
192.168.0.1 wireless LAN
Workstation
Mobile devices with
NetMotion Mobility Client
Server with:
- Firewall rules
- NAT (many-to-one)
- Router
192.168.0.2 wireless LAN
Workstation
NetMotion
Mobility Server
Wired network
123.111.22.x
123.111.23.x
Access point
Private Wired Network
Public Network

Figure 5C

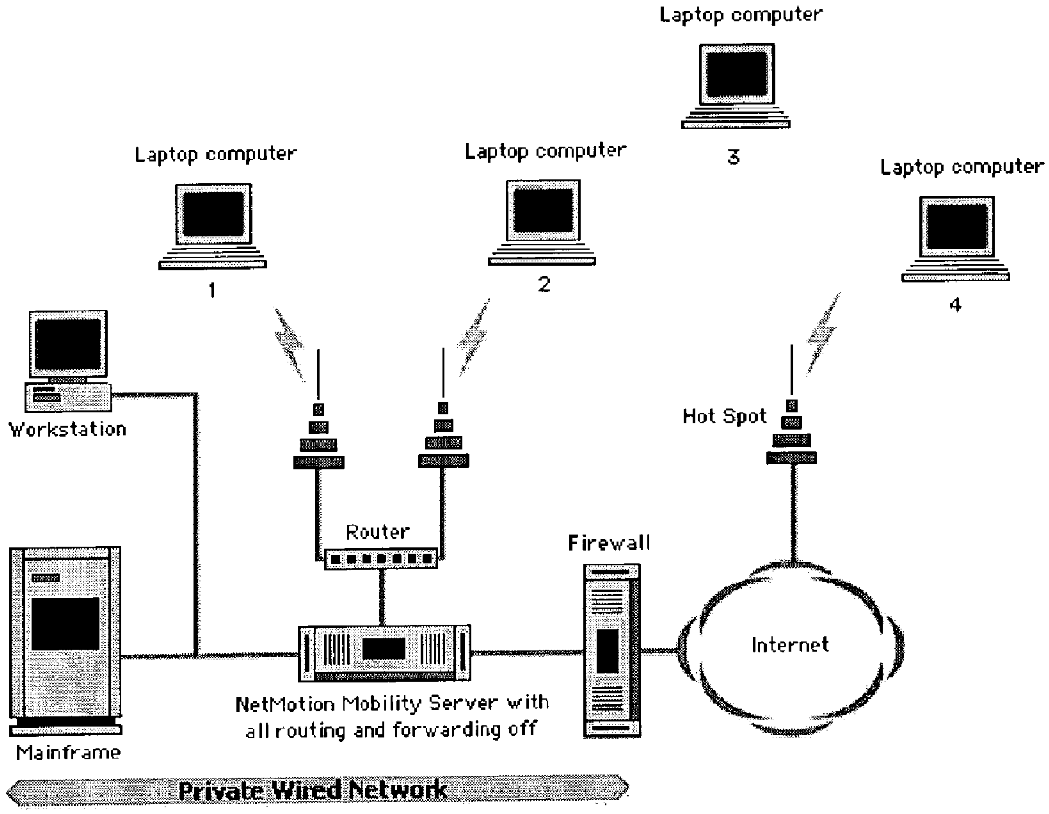

Laptop computer
3
Laptop computer
1
Laptop computer
2
Laptop computer
4
Workstation
Hot Spot
Router
Firewall
Internet
NetMotion Mobility Server with all routing and forwarding off
Mainframe
Private Wired Network

Figure 5D

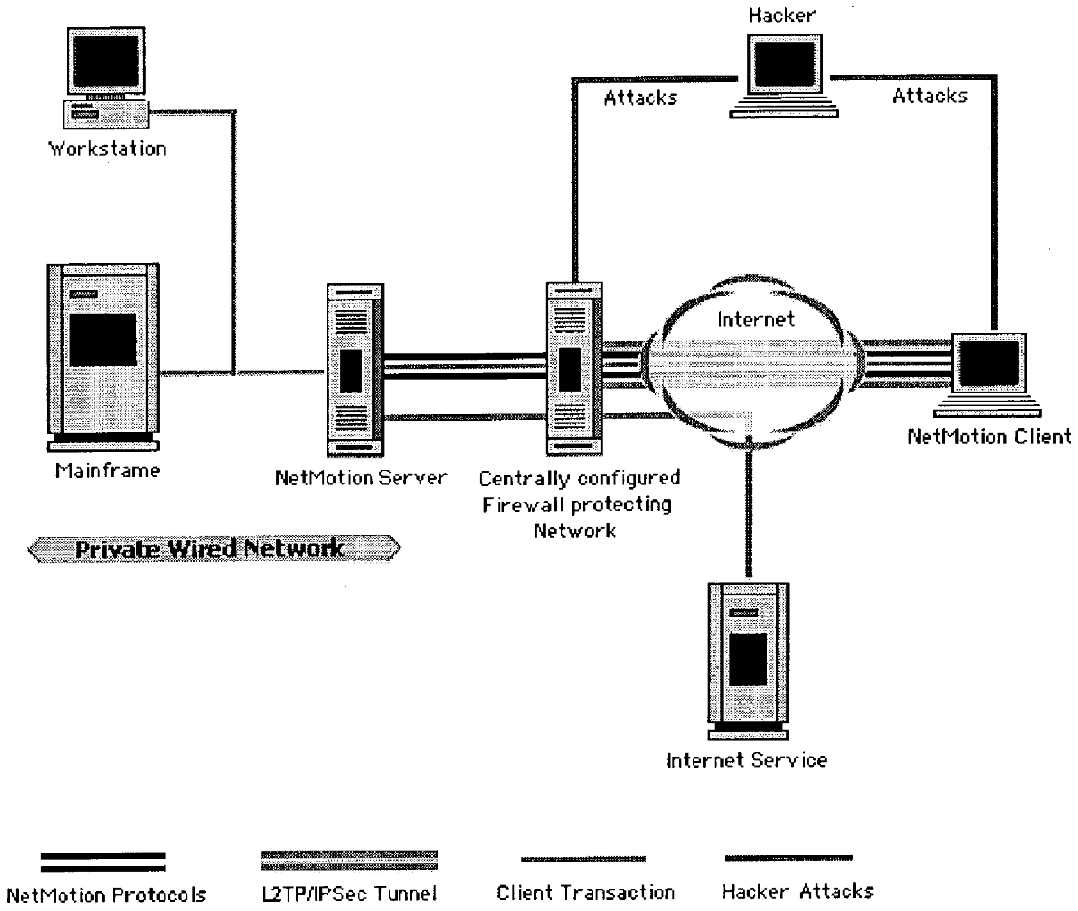

Hacker
Attacks
Attacks
Workstation
Internet
Mainframe
NetMotion Server
Centrally configured Firewall protecting Network
NetMotion Client
Private Wired Network
Internet Service
NetMotion Protocols
L2TP/IPSec Tunnel
Client Transaction
Hacker Attacks

Figure 5E

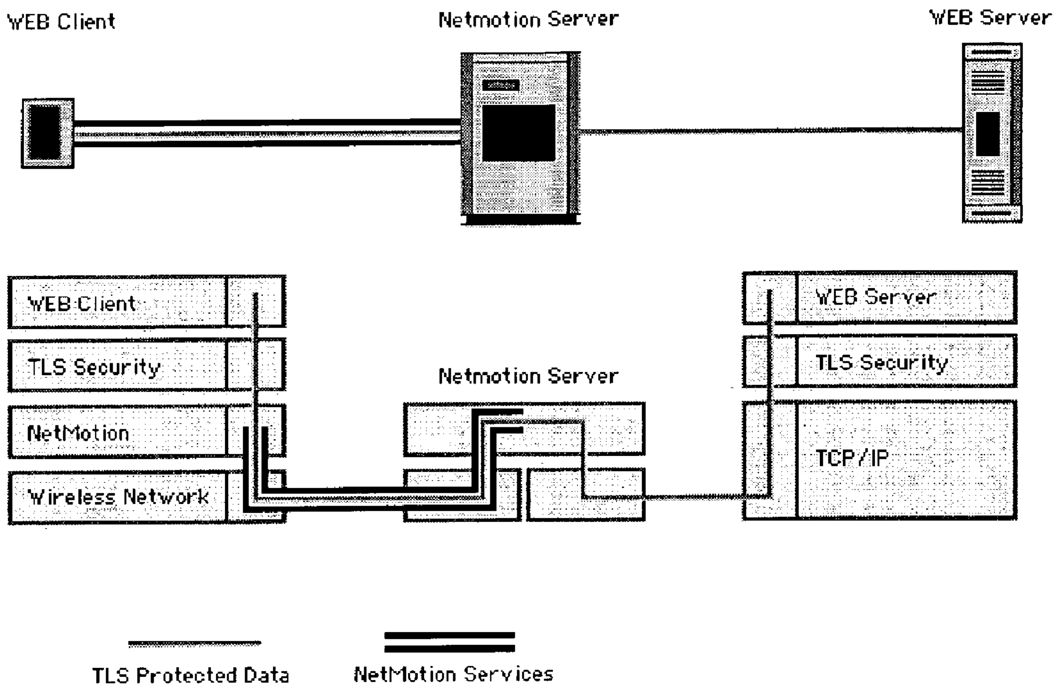

WEB Client
Netmotion Server
WEB Server
WEB Client
TLS Security
NetMotion
Wireless Network
Netmotion Server
WEB Server
TLS Security
TCP/IP
TLS Protected Data
NetMotion Services

Figure 5F

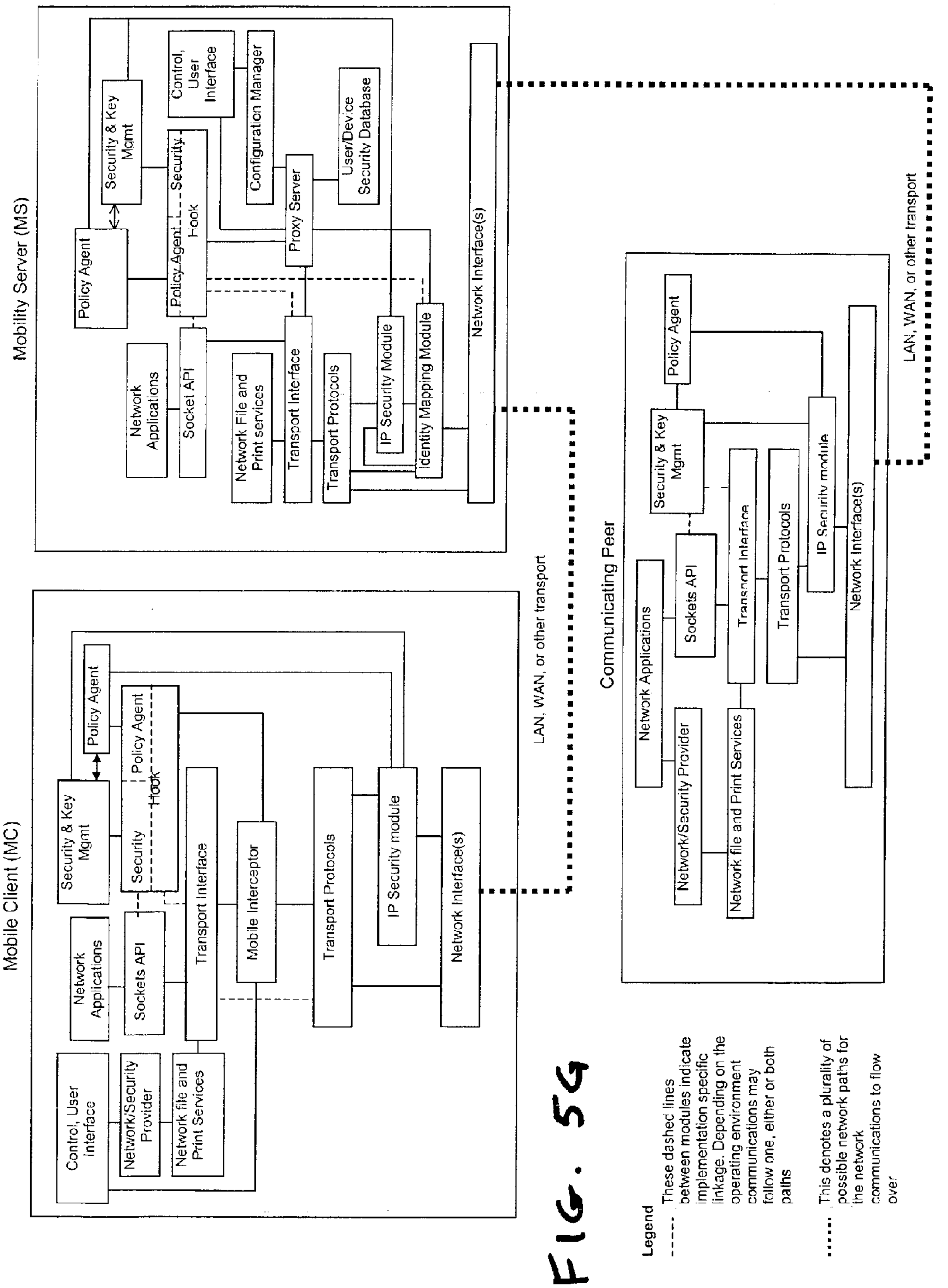

Mobility Server (MS)
Policy Agent
Security & Key Mgmt
Policy Agent
Security Hook
Control, User Interface
Configuration Manager
Proxy Server
User/Device Security Database
Network Applications
Socket API
Network File and Print services
Transport Interface
Transport Protocols
IP Security Module
Identity Mapping Module
Network Interface(s)
LAN, WAN, or other transport
LAN, WAN, or other transport
Mobile Client (MC)
Control, User Interface
Network/Security Provider
Network file and Print Services
Network Applications
Sockets API
Security & Key Mgmt
Security
Policy Agent
Hook
Policy Agent
Transport Interface
Mobile Interceptor
Transport Protocols
IP Security module
Network Interface(s)
Communicating Peer
Network Applications
Network/Security Provider
Sockets API
Security & Key Mgmt
Policy Agent
Transport Interface
Network file and Print Services
Transport Protocols
IP Security module
Network Interface(s)
Legend
These dashed lines between modules indicate implementation specific linkage. Depending on the operating environment communications may follow one, either or both paths
This denotes a plurality of possible network paths for the network communications to flow over

FIG. 5G

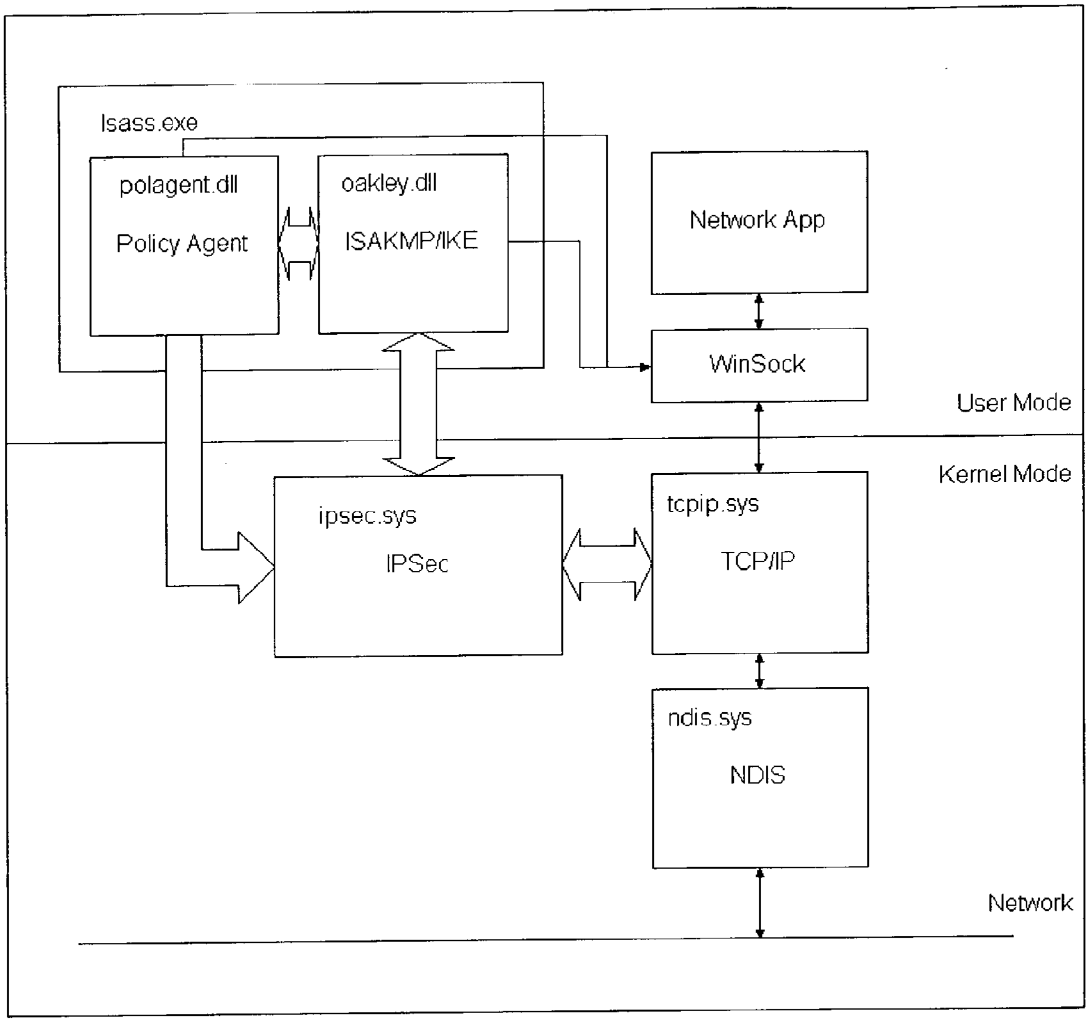


Figure 6 – Example IPSec Operating System Security Architecture
(Prior Art)

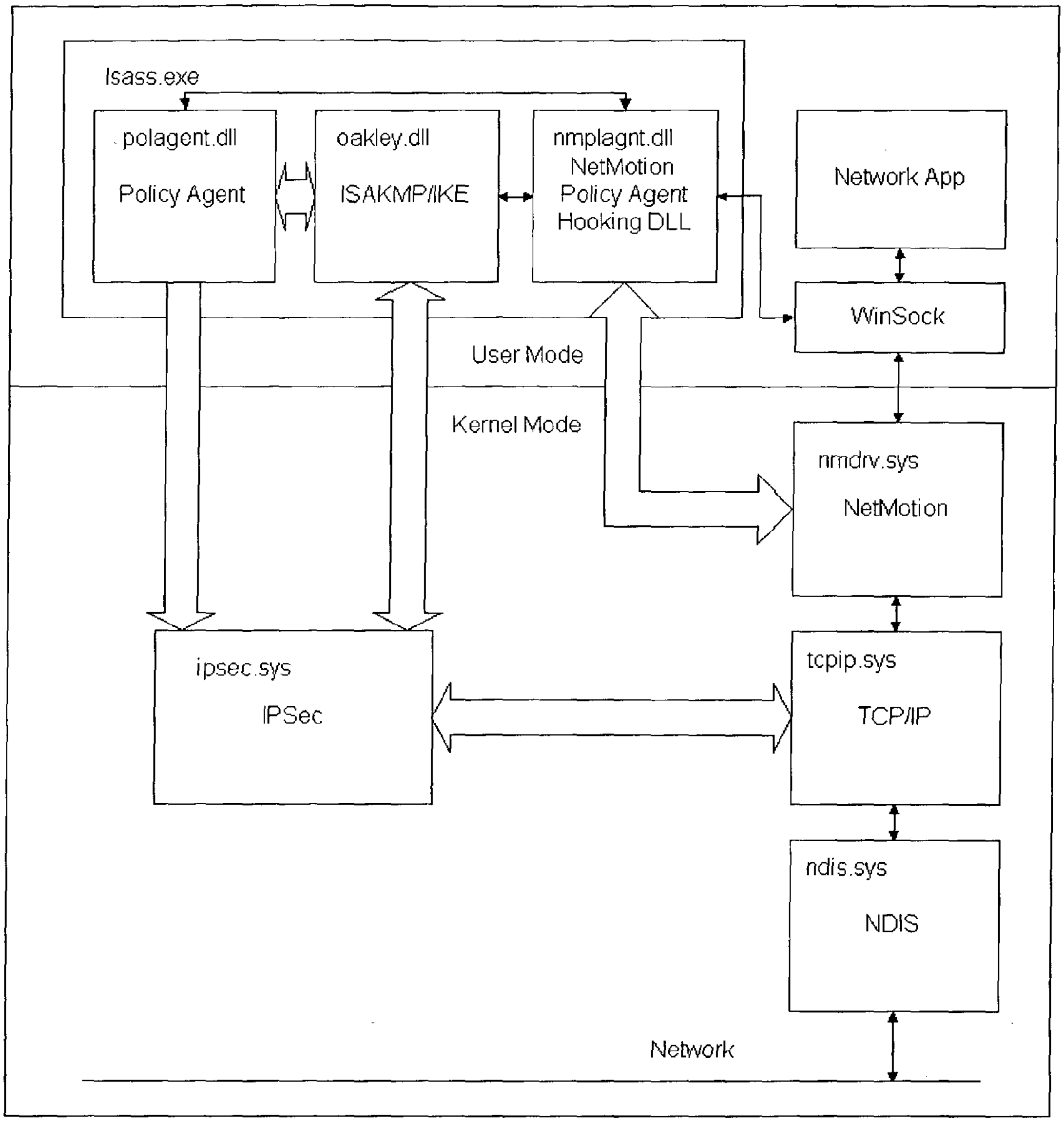
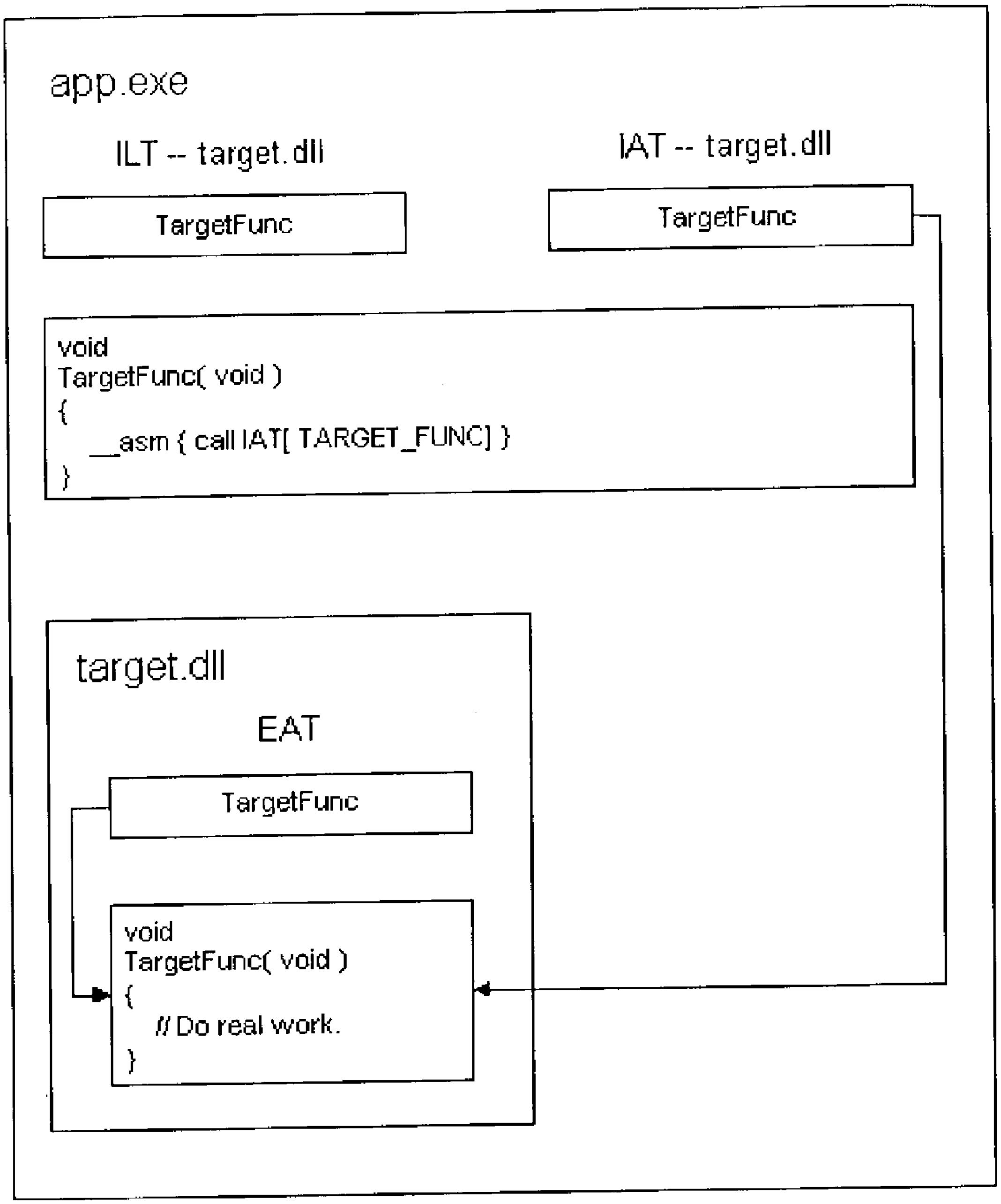


Figure 7 – Example Client Architccture



```
void
TargetFunc( void )
{
   __asm { call IAT[ TARGET_FUNC] }
}
```

```
void
TargetFunc( void )
{
   // Do real work.
}
```

Figure 8 – Example Run-time Linking Sample

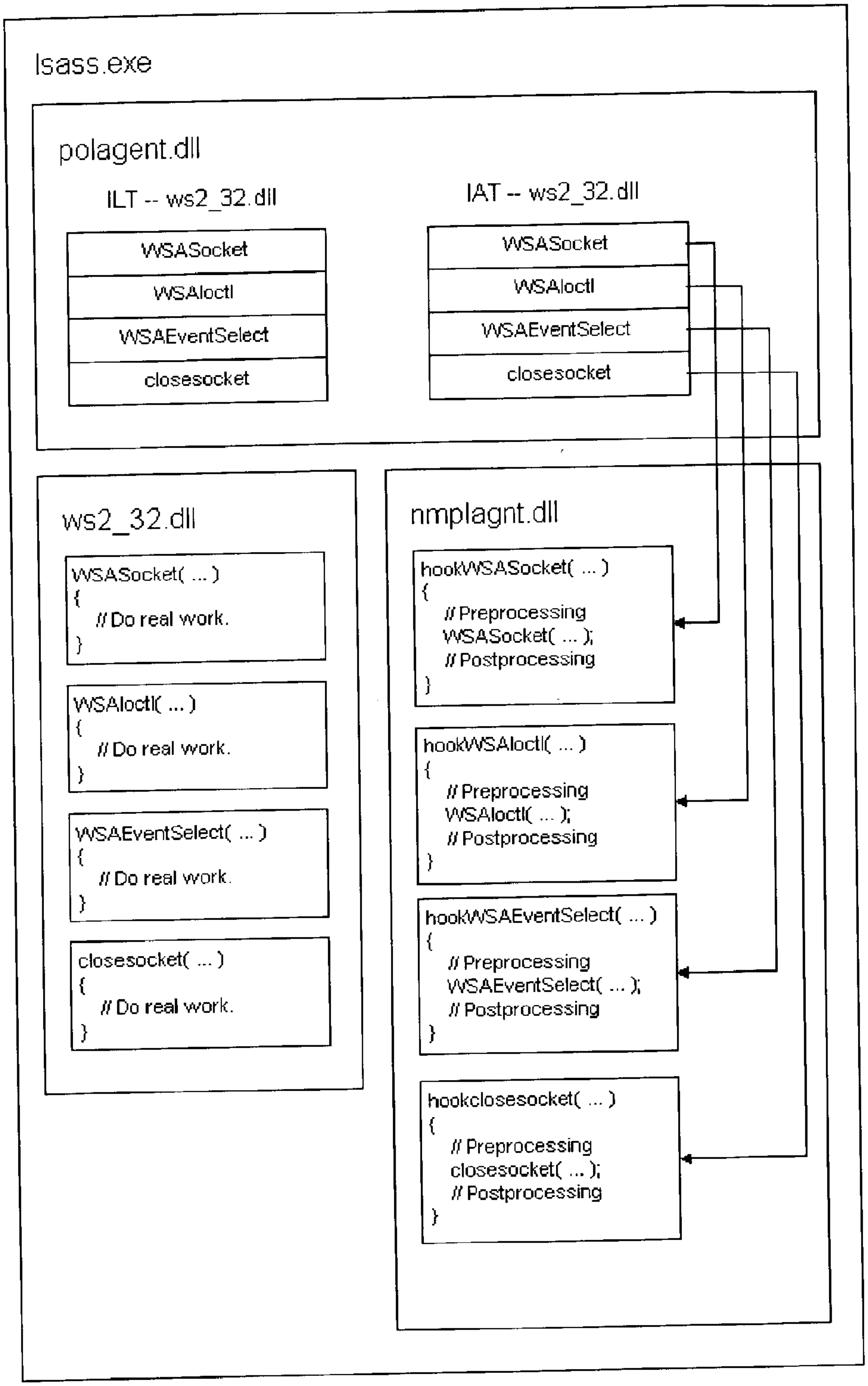

lsass.exe
polagent.dll
ILT -- ws2_32.dll
WSASocket
WSAIoctl
WSAEventSelect
closesocket
IAT -- ws2_32.dll
WSASocket
WSAIoctl
WSAEventSelect
closesocket
ws2_32.dll
WSASocket(...) { // Do real work. }
WSAIoctl(...) { // Do real work. }
WSAEventSelect(...) { // Do real work. }
closesocket(...) { // Do real work. }
nmplagnt.dll
hookWSASocket(...) { // Preprocessing WSASocket(...); // Postprocessing }
hookWSAIoctl(...) { // Preprocessing WSAIoctl(...); // Postprocessing }
hookWSAEventSelect(...) { // Preprocessing WSAEventSelect(...); // Postprocessing }
hookclosesocket(...) { // Preprocessing closesocket(...); // Postprocessing }

Figure 9 – Example Client Policy Agent Hooking

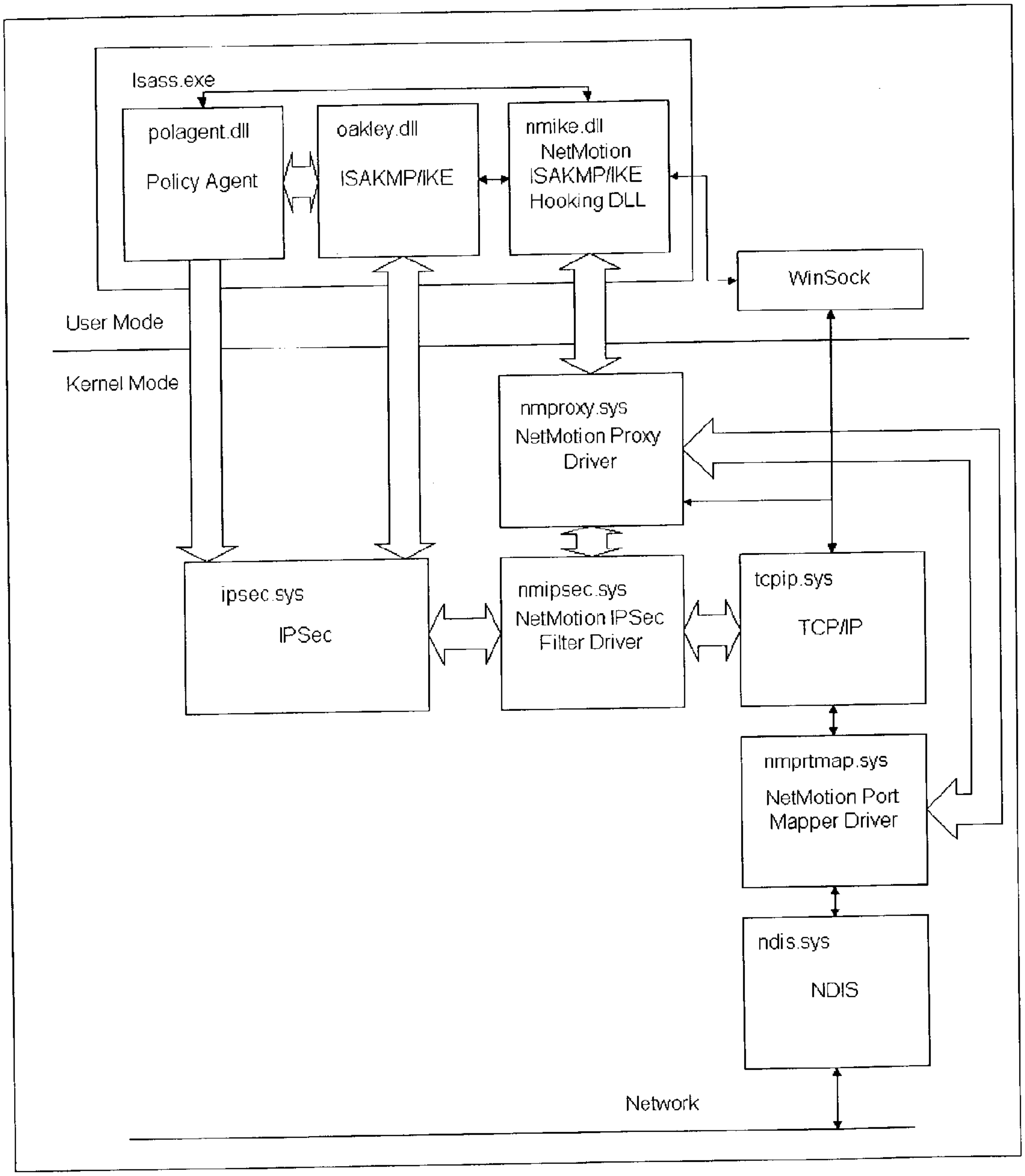


Figure 10 – Example Server Architecture

METHOD AND APPARATUS FOR PROVIDING SECURE CONNECTIVITY IN MOBILE AND OTHER INTERMITTENT COMPUTING ENVIRONMENTS

CROSS-REFERENCE TO RELATED APPLICATIONS

[0001] This application claims the benefit of priority from the following copending commonly-assigned related U.S. patent applications:

> **[0002]** U.S. Provisional Application No. 60/347,243, filed Jan. 14, 2002 (Attorney Docket 3978-9);
>
> **[0003]** U.S. Provisional Application Serial No. 60/274,615 filed Mar. 12, 2001, entitled "Method And Apparatus For Providing Mobile and Other Intermittent Connectivity In A Computing Environment" (Attorney Docket 3978-6);
>
> **[0004]** U.S. patent application Ser. No. 09/330,310 filed Jun. 11, 1999, entitled "Method And Apparatus For Providing Mobile and Other Intermittent Connectivity In A Computing Environment" (Attorney Docket 3978-3);
>
> **[0005]** U.S. patent application Ser. No. 09/660,500 filed Sep. 12, 2000, entitled "Method And Apparatus For Providing Mobile and Other Intermittent Connectivity In A Computing Environment" (Attorney Docket 3978-2); and
>
> **[0006]** PCT International Application Number PCT/US01/28391 filed Sep. 12, 2001, entitled "Method And Apparatus For Providing Mobile And Other Intermittent Connectivity In A Computing Environment" (Attorney Docket 3978-7).

[0007] All of the above-identified documents are incorporated herein by

STATEMENT REGARDING FEDERALLY SPONSORED RESEARCH OR DEVELOPMENT

[0008] Not applicable.

BACKGROUND AND SUMMARY OF THE INVENTION

[0009] Wireless networks have become very popular. Students are accessing course information from the college's computer network while sitting in lecture hall or enjoying the outdoors in the middle of the college campus. Doctors are maintaining computing connectivity with the hospital computer network while making their rounds. Office workers can continue to work on documents and access their email as they move from their office to a conference room. Laptop or PDA users in conference centers, hotels, airports and coffee houses can surf the web and access email and other applications over the Internet. Home users are using wireless networks to eliminate the need to run cables.

[0010] Wireless connectivity provides great flexibility but also presents security risks. Information transmitted through a cable or other wired network is generally secure because one must tap into the cable in order to access the transmission. However, information transmitted wirelessly can be received by anyone with a wireless receiver who is in range. Security risks may not present much of a problem to students reading course material or to cafe customers surfing the World Wide Web, but they present major concerns to businesses and professionals as well as their clients, customers and patients.

[0011] Generally, wired and wireless computing worlds operate under very different paradigms. The wired world assumes a fixed address and a constant connection with high bandwidth. A wireless environment, in contrast, exhibits intermittent connections and has higher error rates over what is usually a narrower bandwidth. As a result, applications and messaging protocols designed for the wired world don't always work in a wireless environment. However, the wireless expectations of end users are set by the performance and behaviors of their wired networks. Meeting these expectations creates a significant challenge to those who design and develop wireless networking architectures, software and devices.

[0012] Authenticating users and keeping communications confidential are more problematic in a wireless network than they are in a wired network. Wireless networks generally are subject to much greater varieties of attacks (e.g., man-in-the-middle, eavesdropping, "free rides" and wide area imposed threats) and assumptions that often do not apply to wired networks. For example, in modern network topologies such as wireless networks and Internet-based virtual private networks (VPNs), physical boundaries between public and private networks do not exist. In such networks, whether a user has the necessary permissions to access the system can no longer be assumed based on physical location as with a wired network in a secure facility. Additionally, wireless data is often broadcasted on radio frequencies, which can travel beyond the control of an organization, through walls and ceilings and even out into the parking lot or onto the street. The information the network is carrying is therefore susceptible to eavesdropping. Imagine if vital hospital patient information could be intercepted or even altered by an unauthorized person using a laptop computer in the hospital lobby, or if a corporate spy could learn his competitor's secrets by intercepting wireless transmissions from an office on the floor above or from a car in the parking lot. While tapping into a wired network cable in a secure facility is possible, the chances of this actually happening are less likely than interception of radio transmissions from a wireless network. Further security threats and problems must be faced when users wish to use any of the ever-increasing variety of public wireless networks to access sensitive data and applications.

[0013] Many of the open standards that make it possible for wireless network hardware vendors to create interoperable systems provide some form of security protection. For example, the IEEE 802.11b "Wi-Fi" standard has been widely implemented to provide wireless connectivity for all sorts of computing devices. It provides an optional Wired Equivalent Privacy ("WEP") functionality that has been widely implemented. Various additional wireless related standards attempt to address security problems in wireless networks, including for example:

[0014] Wireless Application Protocol (WAP) and the associated Wired Transport Layer Security (WTLS); and

[0015] Mobile IP.

[0016] However, as explained below and as recognized throughout the industry, so far these standards have not provided a complete, easy-to-implement transparent security solution for mobile computing devices that roam between different networks or subnetworks.

[0017] WAP generally is designed to transmit data over low-bandwidth wireless networks to devices like mobile telephones, pagers, PDA's, and the like. The Wired Transport Layer Security (WTLS) protocol in WAP provides privacy, data integrity and authentication between WAP-based applications. A WAP gateway converts between the WAP protocol and standard web and/or Internet protocols such as HTTP and TCP/IP, and WTLS is used to create a secure, encrypted pipe. One issue with this model is that once the intermediate WAP gateway decrypts the data, it is available in clear text form—presenting an opportunity for the end-to-end security of the system to be compromised. Additionally, WAP has typically not been implemented for high-bandwidth scenarios such as wireless local area network personal computer connectivity.

[0018] WEP (Wired Equivalent Privacy) has the goal of providing a level of privacy that is equivalent to that of an unsecured wired local area network. WEP is an optional part of the IEEE 802.11 standard, but many hardware vendors have implemented WEP. WEP provides some degree of authentication and confidentiality, but also has some drawbacks and limitations.

[0019] To provide authentication and confidentiality, WEP generally relies on a default set of encryption keys that are shared between wireless devices (e.g., laptop computers with wireless LAN adapters) and wireless access points. Using WEP, a client with the correct encryption key can "unlock" the network and communicate with any access point on the wireless network; without the right key, however, the network rejects the link-level connection request. If they are configured to do so, WEP-enabled wireless devices and access points will also encrypt data before transmitting it, and an integrity check ensures that packets are not modified in transit. Without the correct key, the transmitted data cannot be decrypted—preventing other wireless devices from eavesdropping.

[0020] WEP is generally effective to protect the wireless link itself although some industry analysts have questioned the strength of the encryption that WEP currently uses. However, a major limitation of WEP is that the protection it offers does not extend beyond the wireless link itself. WEP generally offers no end-to-end protection once the data has been received by a wireless access point and needs to be forwarded to some other network destination. When data reaches the network access point or gateway, it is unencrypted and unprotected. Some additional security solution must generally be used to provide end-to-end authentication and privacy.

[0021] Mobile IP is another standard that attempts to solve some of the problems of wireless and other intermittently-connected networks. Generally, Mobile IP is a standards based algorithm that enables a mobile device to migrate its network point of attachment across homogeneous and heterogeneous network environments. Briefly, this Internet Standard specifies protocol enhancements that allow routing of Internet Protocol (IP) datagrams (e.g., messages) to mobile nodes in the Internet. See for example Perkins, C., "IP Mobility Support", RFC 2002, October 1996.

[0022] Mobile IP contemplates that each mobile node is always identified by its home address, regardless of its current point of attachment to the Internet. While situated away from its home, a mobile node is also associated with a "care-of" address, which provides information about its current point of attachment to the Internet. The protocol provides for registering the "care-of" address with a home agent. The home agent sends datagrams destined for the mobile node through a "tunnel" to the "care-of" address. After arriving at the end of the "tunnel," each datagram is then delivered to the mobile node.

[0023] While Mobile IP provides useful techniques for remote connectivity, it is not yet widely deployed/implemented. This seems to be due to a variety of factors—at least one of which is that there continues to be some unsolved problems or areas where the Mobile IP standard is lacking and further enhancement or improvement would be desirable. For example, even though security is now fairly widely recognized as being a very important aspect of mobile networking, the security components of Mobile IP are still mostly directed to a limited array of security problems such as redirection attacks.

[0024] Redirection attacks are a very real threat in any mobility system. For example, a redirection attack can occur when a malicious node gives false information to a home agent in a Mobile IP network (e.g., sometimes by simply replaying a previous message). This is similar to someone filing a false "change of address" form with the Post Office so that all your mail goes to someone else's mailbox. The home agent is informed that the mobile node has a new "care-of" address. However, in reality, this new "care-of" address is controlled by the malicious node. After this false registration occurs, all IP datagrams addressed to the mobile node are redirected to the malicious node.

[0025] While Mobile IP provides a mechanism to prevent redirection attacks, there are other significant security threats that need to be addressed before an enterprise can feel comfortable with the security of their wireless network solution. For example, Mobile IP generally does not provide a comprehensive security solution including mobile computing capabilities such as:

[0026] Session resilience/persistence

[0027] Policy management

[0028] Distributed firewall functionality

[0029] Location based services

[0030] Power management

[0031] Other capabilities.

[0032] While much security work has been done by the Internet community to date in the Mobile IP and other contexts, better solutions are still possible and desirable. In particular, there continues to be a need to provide an easy-to-use, comprehensive mobility solution for enterprises and other organizations who wish to add end-to-end security to existing and new infrastructures that make extensive use of existing conventional technology and standards and which support mobility including roaming transparently to applications that may not be "mobile-aware." Some solutions exist, but many of them require changes to existing infrastructure that can be difficult to implement and maintain.

[0033] For example, in terms of the current implementations that do exist, Mobile IP is sometimes implemented as a "bump" in the TCP/IP protocol stack to replace components of the existing operating system environment. An example of such an architecture is shown in prior art **FIG. 1**. In the exemplary illustrative prior art arrangement shown, a Mobile IP module sits below the regular TCP/IP protocol stack components and manages the transitions from one network to another. Generally, using such solution, additions or modifications to existing core network infrastructure entities are needed to facilitate the behavior of nomadic or migratory computing. The need for such modifications makes widespread implementation difficult and causes problems in terms of maintainability and compatibility.

[0034] Another common security solution that enterprises have gravitated toward is something called a Virtual Private Network (VPN). VPNs are common on both wired and wireless networks. Generally, they connect network components and resources through a secure protocol tunnel so that devices connected to separate networks appear to share a common, private backbone. VPN's accomplish this by allowing the user to "tunnel" through the wireless network or other public network in such a way that the "tunnel" participants enjoy at least the same level of confidentiality and features as when they are attached to a private wired network. Before a "tunnel" can be established, cryptographic methods are used to establish and authenticate the identity of the tunnel participants. For the duration of the VPN connection, information traversing the tunnel can be encrypted to provide privacy.

[0035] VPN's provide an end-to-end security overlay for two nodes communicating over an insecure network or networks. VPN functionality at each node supplies additional authentication and privacy in case other network security is breached or does not exist. VPN's have been widely adopted in a variety of network contexts such as for example allowing a user to connect to his or her office local area network via an insecure home Internet connection. Such solutions can offer strong encryption such as the AES (Advanced Encryption Standard), compression, and link optimizations to reduce protocol chattiness. However, many or most VPNs do not let users roam between subnets or networks without "breaking" the secure tunnel. Also, many or most VPNs do not permit transport, security and application sessions to remain established during roaming. Another potential stumbling block is conventional operating systems—not all of which are compatible with the protection of existing wireless VPNs.

[0036] To address some of the roaming issue, as previously mentioned, standards efforts have defined Mobile IP. However, Mobile IP, for example, operates at the network layer and therefore does not generally provide for session persistence/resilience. If the mobile node is out of range or suspended for a reasonably short period of time, it is likely that established network sessions will be dropped. This can present severe problems in terms of usability and productivity. Session persistence is desirable since it lets the user keep the established session and VPN tunnel connected—even if a coverage hole is entered during an application transaction. Industry analysts and the Wireless Ethernet Compatibility Alliance recommend that enterprises deploy VPN technology, which directly addresses the security problem, and also provides advanced features like network and subnet roaming, session persistence for intermittent connections, and battery life management for mobile devices. However, VPN solutions should desirably support standard security encryption algorithms and wireless optimizations suitable for today's smaller wireless devices, and should desirably also require no or minimal modification to existing infrastructure.

[0037] One standards-based security architecture and protocol approach that has been adopted for providing end-to-end secure communications is called "Internet Security Protocol" ("IPSec"). IPSec is a collection of open standards developed by the Internet Engineering Task Force (IETF) to secure communications over public and private networks. See for example:

> **[0038]** RFC 1827 "IP Encapsulating Security Payload (ESP)" R. Atkinson (August 1995);
>
> **[0039]** RFC 1826 "IP Authentication Header" R. Atkinson. (August 1995); and
>
> **[0040]** RFC 1825 "Security Architecture for the Internet Protocol" R. Atkinson (August 1995).

[0041] Briefly, IPSec is a framework for ensuring private, secure communications over Internet Protocol (IP) networks, through the use of cryptographic security services. The IPSec suite of cryptography-based protection services and security protocols provides computer-level user and message authentication, as well as data encryption, data integrity checks, and message confidentiality. IPSec capabilities include cryptographic key exchange and management, message header authentication, hash message authentication, an encapsulating security payload protocol, Triple Data Encryption, the Advanced Encryption Standard, and other features. In more detail, IPSec provides a transport mode that encrypts message payload, and also provides a tunnel mode that encrypts the payload, the header and the routing information for each message. To reduce overhead, IPSec uses policy-based administration. IPSec policies, rather than application programming interfaces (APIs), are used to configure IPSec security services. The policies provide variable levels of protection for most traffic types in most existing networks. One can configure IPSec policies to meet the security requirements of a computer, application, organizational unit, domain, site, or global enterprise based on IP address and/or port number.

[0042] IPSec is commonly used in firewalls, authentication products and VPNs. Additionally, Microsoft has implemented IPSec as part of its Windows 2000 and Windows XP operating system. IPSec's tunnel mode is especially useful in creating secure end-to-end VPNs. IPSec VPNs based on public key cryptography provide secure end-to-end message authentication and privacy. IPSec endpoints act as databases that manage and distribute cryptographic keys and security associations. Properly implemented, IPSec can provide private channels for exchanging vulnerable data such as email, file downloads, news feeds, medical records, multimedia, or any other type of information.

[0043] One might initially expect that it should be relatively straightforward to add a security algorithm such as the standards-based IPSec security algorithm to Mobile IP or other mobility protocol. For example, layering each of the entities in the fashion such as that shown in prior art **FIG. 2** would seem to allow for security in an environment where the mobile node's IP address never needs to change. Thus, the IPSec security association between the mobile node and its ultimate peer could be preserved across network segment boundaries, and end-to-end security would also preserved. However, combining the Mobile IP and IPSec algorithms in this manner can present its own set of problems.

[0044] For example, when the mobile node has roamed to a foreign network and is communicating with its ultimate peer, it is possible that packets generated by the mobile node may be discarded by a policy enforcement entity such as a firewall. This can be due to common practice known as ingress filtering rules. Many firewalls discard packets generated by mobile nodes using their home addresses (internal network identity) and received on an externally facing network interface in defense of the network. This discarding process is intended to protect the network secured by the firewall from being attacked. Ingress filtering has the effect of forcing the tunneling of Mobile IP frames in both directions. See for example RFC 2356 Sun's SKIP Firewall Traversal for Mobile IP. G. Montenegro, V. Gupta. (June 1998).

[0045] Additionally, it is becoming general practice in the industry to require that an IPSec security session be established between the foreign agent and the externally facing policy enforcement equipment (e.g. firewall) before allowing packets to traverse between the external and internal network interconnection (a.k.a. VPN). If the foreign agent is co-located with the mobile node, this can become a cumbersome operation. As exemplary **FIG. 3** depicts, yet another level of network protocol enveloping could be used to meet possibly required security policies to allow network traffic to flow between the mobile node to the foreign agent through the policy enforcement equipment (e.g. firewall) to the home agent and then to the other communications end point (i.e. ultimate peer). However, this adds substantial additional overhead due to the additional encapsulation. Furthermore, if the foreign agent entity is not co-located with the mobile node (or up to policy restrictions on the newly attached network), a specific foreign agent may need to be used for these communications, and credential information must somehow be shared between the foreign agent and the terminus of the first (outer) IPSec session. A drawback to this methodology is that it can increase the security risk by sharing credential information with a network entity that may not be directly under user or corporate administrative control.

[0046] What is needed is a solution to these problems providing security, network roaming, and session persistence over conventional information communications networks including but not limited to standard IP based networks without requiring modification to existing network applications. Additionally, it would be useful if such a solution did not require the deployment of Mobile IP or any additional infrastructure such as a foreign agent when visiting a remote network, and the functionality can be transparent to networked applications so they do not need to be modified either.

[0047] This invention solves this problem by transparently providing secure, persistent, roamable IP-based communications using conventional technologies such as IPSec, Microsoft or other operating system security functionality while avoiding the commonly experienced ingress filtering problems. And unlike at least some implementations of Mobile IP, few if any changes are necessary to the underlying network infrastructure.

[0048] Generally, one preferred exemplary non-limiting embodiment provides Mobility Client (MC) functionality that virtualizes the underlying network. Applications running on the mobility client see at least one consistent virtual network identity (e.g. IP address). When an application on the mobility client makes a network request, the mobility client intercepts the request and marshals the request to a Mobility Server (MS) that supports security such as IPSEC. The mobility server unwraps the request and places it on the network as though the server were the client—thus acting as a proxy for the client.

[0049] The reverse also occurs in the exemplary embodiment. When a peer host sends a packet to the mobility client's virtual network identity, the packet is first received by the mobility server and is then transferred to the mobility client. The mobility server maintains a stable point of communication for the peer hosts while the mobility client is free to roam among networks as well as suspend or roam out of range of any network. When the mobility client is out of range, the mobility server keeps the mobility client's sessions alive and queues requests for the mobility client. When the mobility client is once again reachable, the mobility server and client transfer any queued data and communication can resume where it left off.

[0050] Preferred exemplary non-limiting implementations thus offer wireless optimizations and network and application session persistence in the context of a secure VPN or other connection. Wireless optimizations allow data to be transmitted as efficiently as possible to make maximal use of existing bandwidth. For example, the system can be used to switch automatically to the fastest bandwidth network connection when multiple connections (Wi-Fi and GPRS, for example) are active. Network session persistence means that users don't have to repeat the login process when they move from one IP subnet to another, or when they go out of range of the network and return. Exemplary implementations automatically re-authenticate the connection every time users roam, without need for user intervention. Application session persistence means that standard network applications remain connected to their peers, preventing the loss of valuable user time and data. Such optimizations and persistence is provided in the context of a security architecture providing end-to-end security for authentication and privacy.

[0051] In one illustrative embodiment, before data is transported between the network and a mobility client, the network ensures that the end user has the required permissions. A user establishes her identity by logging in to the mobility client using a conventional (e.g., Windows) domain user name and password. Using the conventional domain credentials allows for a single sign-on process and requires no additional authentication tables or other infrastructure additions. Single sign-on also gives users access to other domain resources such as file system shares. Once a user has been authenticated, a communications path is established for transporting application data. Any number of different protocols (e.g., Common Internet File System, Radius, other) can be used for user authentication. Using certain of these protocols, a mobility server can act as a Network Access Server to secure an initial access negotiation which establishes the user's user name and password using conventional protocols such as EAP-MD5, LEAP, or other protocol. Unlike some wireless protocols, such authentication in the exemplary non-limiting implementations provides user-specific passwords that can be used for policy management allowing access and resource allocation on a user basis.

[0052] Significantly, exemplary non-limiting implementations can be easily integrated with IPSEC or other security features in conventional operating systems such as for example Windows NT and Windows 2000. This allows access to conventional VPN and/or other proven-secure connection technology. IPSec policies can be assigned through the group policy feature of Active Directory, for example. This allows IPSec policy to be assigned at the domain or organizational level—reducing the administrative overhead of configuring each computer individually. An on-demand security negotiation and automatic key management service can also be provided using the conventional IETF-defined Internet Key Exchange (IKE) as specified in Internet RFC 2409. Such exemplary implementations can provide IEFT standards-based authentication methods to establish trust relationships between computers using public key cryptography based certificates and/or passwords such as preshared keys. Integration with conventional standards-based security features such as public key infrastructure gives access to a variety of security solutions including secure mail, secure web sites, secure web communications, smart card logon processes, IPSec client authentication, and others.

[0053] Illustrative exemplary embodiments can be cognizant of changes in network identity, and can selectively manage transition in network connectivity, possibly resulting in the termination and/or (re)instantiation of IPSec security sessions between communicating entities over at least one of a plurality of network interfaces. Exemplary illustrative embodiments also provide for the central management, distribution, and/or execution of policy rules for the establishment and/or termination of IP security sessions as well as other parameters governing the behavior for granting, denying and/or delaying the consumption of network resources.

[0054] Illustrative non-limiting advantageous features include:

[0055] Roamable IPSec allows IPSec tunnel to automatically roam with mobile computing devices wherever they go—based on recognized IPSec security standard, Roamable IPSec enables seamless roaming across any physical or electronic boundary with the authentication, integrity and encryption of IPSec, to provide a standards-based solution allowing mobile and remote users with VPN-level security and encryption in an IPSec tunnel that seamlessly roams with wireless users wherever they go and however they access their enterprise data.

[0056] Detecting when a change in network point of attachment, an interruption of network connectivity, a roam to a different network or other subnetworks, a mobile client's identify, or other discontinuity has occurred on the mobile client and (re)instantiating an IP Security session while maintaining network application sessions—all in a manner that is transparent to the networked application.

[0057] Transparently and selectively injecting computer instructions and redirecting the execution path of at least one software or other component based for example on process name to achieve additional level(s) of functionality while maintaining binary compatibility with operating system components, transport protocol engines, and/or applications.

[0058] Selectively but transparently virtualizing at least one network interface for applications and operating system components—shielding them from the characteristics of mobile computing while allowing other components to remain cognizant of interruptions in connectivity and changes in network point of attachment.

[0059] Selectively virtualizing at least one network interface for network applications and operating system components thus shielding them from adverse events that may disturb communications such as changes in network point of attachment and/or periods of disconnectedness.

[0060] Allowing the establishment of multiple IP Security sessions over one or more network interfaces associated with at least one network point of attachment and allowing network application communications to simultaneously flow over any or all of the multiple IP security sessions and correctly multiplex/demultiplex these distributed communication flows into corresponding higher layer communications sessions.

[0061] Applying policy rules to selectively allow, deny, and/or delay the flow of network communications over at least one of a plurality of IP Security sessions.

[0062] Centrally managing and/or distributing policy regarding the establishment of IP Security sessions from a central authority.

[0063] An "Add session" concept—during the proxying of communications for a mobile client, the mobility server can instantiate at least one of a possible plurality of IP Security sessions between a mobility server and an ultimate peer on behalf of a mobility client.

[0064] Establishing and maintaining IP Security sessions between the Mobility Server and ultimate communications peer, even during periods when the mobility client is unreachable.

[0065] Automatically terminating IP Security sessions between the mobility server and ultimate communications peer, based on, but not limited to link inactivity, application session inactivity, or termination of a communications end point.

[0066] Associating at least one IP security session between the mobility server and ultimate peer and mobility client and mobility server regardless of the current mobility client network identities.

[0067] Transparently injecting computer instructions and redirecting the execution path to achieve additional level(s) of functionality while maintaining binary compatibility with operating system components, transport protocol engines, and applications.

[0068] Allowing establishment of at least one of a plurality of IP security sessions over a plurality of network interfaces associated with at least one network point of attachment and allowing network application communications to simultaneously flow over at least one of a plurality of IP Security sessions and correctly multiplex/demultiplex these distributed communication flows back into corresponding higher layer communications sessions.

[0069] Selectively virtualizing at least one network interface for network applications and operating system components thus shielding them any adverse events that may disrupt communications such as changes in network point of attachment and/or periods of disconnectedness.

[0070] Centrally managing and distributing policy rules regarding the establishment and/or termination of IP security sessions for mobile clients and/or mobility servers from a central authority.

[0071] A mobility security solution that starts at the mobile device and provides both secure user authentication and, when needed, secure data encryption.

[0072] A mobility security solution that voids the need for single-vendor solutions not based on industry-wide, open and other standards.

[0073] Secure VPN that is extendable to a variety of different public data networks having different configurations (e.g., Wi-Fi network hotspot, wide-area wireless solutions such as CDPD or GPRS, etc.) dynamically controllable by the network administrator

[0074] A mobility security solution that works with a wide variety of different computing devices of different configurations running different operating systems.

[0075] Allows users to suspend and reestablish secure sessions to conserve battery power while maintaining network application sessions.

[0076] Provides a secure solution in a wireless topology that has dead spots and coverage holes.

[0077] No need to develop custom mobile applications or use mobile libraries to get applications to work in a mobile environment.

[0078] Secure transport and application session persistence

[0079] works within existing network security so the network is not compromised.

[0080] compatible with any of a variety of conventional security protocols including for example RADIUS, Kerberos, Public Key Infrastructure (PKI), and Internet Security Protocol (IPSec).

[0081] The computing environment and the applications do not need to change mobility is there to use but its use is transparent to the user and to the applications.

[0082] Since all or nearly all applications run unmodified, neither re-development nor user re-training is required.

[0083] Automatic regeneration of user-session keys at a customized interval.

[0084] Continuous, secure connection ensuring data integrity between wired and wireless data networks.

[0085] Enterprises running VPNs (e.g., PPTP, L2TP/IPSec, IPSec, Nortel, Cisco, other) can use these techniques to add wireless optimization, session persistence and additional security for mobile workers.

[0086] Seamlessly integrates into enterprises where LEAP or other access point authentication security is deployed to add optimized roamable security and encryption.

BRIEF DESCRIPTION OF THE DRAWINGS

[0087] These and other features and advantages may be better and more completely understood by referring to the following detailed description of exemplary non-limiting illustrative embodiments in conjunction with drawings, of which:

[0088] **FIG. 1** shows an exemplary illustrative prior art mobile IP client architecture;

[0089] **FIG. 2** shows an exemplary illustrative prior art IPSec and Mobile IP architecture;

[0090] **FIG. 3** shows an exemplary illustrative network protocol enveloping that may be used to meet the possible required security policies to allow network traffic to flow between a mobile node to a foreign agent through policy enforcement equipment (e.g. firewall) to the home agent and then to another communications end point;

[0091] **FIG. 4** shows an example mobility architecture in accordance with a presently preferred exemplary illustrative non-limiting embodiment of the present invention;

[0092] FIGS. **5** & **5A-5**F show illustrative usage scenarios;

[0093] **FIG. 5G** shows an exemplary client-server architecture;

[0094] **FIG. 6** shows an example simplified prior art operating system security architecture;

[0095] **FIG. 7** shows the example illustrative **FIG. 6** architecture modified to provide secure transparent illustrative mobility functionality;

[0096] **FIG. 8** shows an example illustrative run time linking sample;

[0097] **FIG. 9** shows an example illustrative client policy agent hooking; and

[0098] **FIG. 10** shows an example illustrative server architecture.

DETAILED DESCRIPTION OF EXEMPLARY NON-LIMITING EMBODIMENTS

[0099] **FIG. 4** shows an exemplary overall illustrative non-limiting mobility architecture. The example mobility architecture includes a mobility client (MC) and a mobility server (MS). The mobility client may be, for example, any sort of computing device such as a laptop, a palm top, a Pocket PC, a cellular telephone, a desktop computer, or any of a variety of other appliances having remote connectivity capabilities. In one exemplary embodiment, mobility client MC comprises a computing-capable platform that runs the Microsoft Windows 2000/XP operating system having security (for example, IPSec) functionality but other implementations are also possible. The system shown is scaleable and can accommodate any number of mobility clients and mobility servers.

[0100] In one exemplary embodiment, mobility client MC may be coupled to a network such as the Internet, a corporate LAN or WAN, an Intranet, or any other computer network. Such coupling can be wirelessly via a radio communications link such as for example a cellular telephone network or any other wireless radio or other communications link. In some embodiments, mobility client MC may be intermittently coupled to the network. The system shown is not, however, limited to wireless connectivity—wired connectivity can also be supported for example in the context of computing devices that are intermittently connected to a wired network. The wireless or other connectivity can be in the context of a local area network, a wide area network, or other network.

[0101] In the exemplary embodiment, mobility client MC communicates with the network using Internet Protocol (IP) or other suitable protocol over at least one of a plurality of possible network interfaces. In the illustrative embodiment, mobility server (MS) is also connected to the network over at least one of a plurality of possible network interfaces. The mobility server MS may communicate with one or more peers or other computing devices. The exemplary **FIG. 4** architecture allows mobility client MC to securely communicate with the peer hosts via the communications link, the network and/or the mobility server MS.

[0102] In more detail, the **FIG. 4** mobility server maintains the state of each mobile device and handles the session management required to maintain continuous connections to network applications. When a mobile device becomes unreachable because it suspends, moves out of coverage or changes its "point of presence" address, the mobility server maintains the connection to the network host by acknowledging receipt of data and queuing requests.

[0103] The exemplary mobility server also manages network addresses for the mobile devices. Each device running on the mobile device has a virtual address on the network and a point of presence address. A standard protocol (e.g., DHCP) or static assignment determines the virtual address. While the point of presence address of a mobile device will change when the device moves from one subnet to another (the virtual address stays constant while the connections are active).

[0104] This illustrative arrangement works with standard transport protocols such as TCP/IP—intelligence on the mobile device and the mobility server assures that an application running on the mobile device remains in sync with its server.

[0105] The mobility server also provides centralized system management through console applications and exhaustive metrics. A system administrator can use these tools to configure and manage remote connections, troubleshoot problems, and conduct traffic studies.

[0106] The mobility server also, in the exemplary embodiment, manages the security of data that passes between it and the mobile devices on the public airways or on a wireline network. The server provides a firewall function by giving only authenticated devices access to the network. The mobility server can also certify and optionally encrypt all communications between the server and the mobile device. Tight integration with Active Directory or other directory/name service provides centralized user policy management for security.

[0107] The **FIG. 4** architecture can be applied in any or all of a large and varying number of situations including but not limited to the exemplary situations shown in **FIG. 5** (for brevity and clarity sake, example embodiments are described using a single network point of attachment but it will be appreciated and understood that the current invention is not to be limited to such scope and application):

> [0108] At example location number one shown in **FIG. 5** and see also **FIG. 5A**, the mobility client (depicted as a laptop computer for purposes of illustration) is shown inside a corporate or other firewall, and is shown connected to a wireless LAN (WLAN) having an access point. In this example, a private wireless network is connected to a wireline network through the mobility server. All application traffic generated on or destined for the wireless network is secured, and no other network traffic is bridged or routed to the wireless network. Using standard firewall features found in the operating system, the system can be further configured to allow only mobility traffic to be processed by the mobility server on the wireless network. In this example, the mobility client is authenticated to the mobility server. Packets flow normally between the mobility client and the mobility server, and the communication channel between the mobility client and the mobility server is protected using the conventional IPSec security protocol.
>
> [0109] At example location number two shown in **FIG. 5**, the mobility client has moved into a deadspot and lost connectivity with the network. The mobility server maintains the mobility client's network applications sessions during this time. Had the mobile client been using Mobile IP instead of the exemplary embodiment herein, the client's sessions could have been dropped because Mobile IP does not offer session persistence.
>
> [0110] At example location number three, the mobility client has moved back into range of the corporate network on a different subnet. The mobility client acquires a new point-of-presence (POP) address on the new subnet, negotiates a new secure channel back to the mobility server using IPSec, reauthenti-

cates with the mobility server, and resumes the previously suspended network sessions without intervention from the user and without restarting the applications. This process is transparent to the mobile applications and to the application server.

[0111] At example location numbers four and five shown in **FIG. 5** and see also **FIGS. 5B & 5C**, the mobility client has left the corporate network and roamed into range of public networks. For example, the mobile client at location **4** shown in **FIG. 5** is shown in range of a conventional Wireless Wide Area Network (WWAN) wireless tower, and the mobile client at location **5** shown in **FIG. 5** is shown in range of a Wi-Fl or other wireless access point "hot spot" such as found in an airport terminal, conference center, coffee house, etc. The wireless technology used for the public network need not be the same as that used inside the enterprise—since the illustrative system provides for secure roaming across heterogeneous networks. The mobility client's traffic must now pass through a corporate or other firewall. The firewall can be configured to pass IPSec traffic intended for the mobility server and/or the mobility client can be configured to use an IPSec session to the firewall. Either solution can be implemented without end-user interaction, although intervention is possible.

[0112] In the **FIG. 5B** example, mobility devices are connected to a diverse, public wide area network. The enterprise is also connected to the public network through a conventional firewall. The firewall is, in the exemplary embodiment, modified to allow mobility connections, specifically to the address of the mobility server. The connections are then protected by conventional security protocols such as IPSec.

[0113] In the **FIG. 5C** example, a private, wired network on a corporate, hospital, or other campus, and a wireless local area network supporting mobile devices connected to it through a conventional firewall. Traffic from the public to the private network that is not destined for the correct port is denied using conventional firewall rules. The firewall rules can specify either the domain ("allow access to 123.111.x:5008") or the addresses of particular mobility servers ("allow access to 123.111.22.3:1002 and 123.111.23.4:5008")—the latter approach being more secure. On a smaller campus, a single, multi-homed mobility server could be used to handle both the wired and wireless LAN traffic. Once a user is authenticated, he or she has access to the wired network. A Network Address Translator (NAT) maybe used to reduce the number of public (routable) IP addresses required. In the example shown in **FIG. 5C, a** many-to-one relationship is provided so that mobile devices can use just one of two IP addresses instead of requiring one address each. Any traffic coming from the wireless LAN access points preferably must satisfy both the firewall rules and be cleared by the mobility server. With encryption enabled, this configuration protects the wired network while offering legitimate wireless users full, secure access to corporate data.

[0114] **FIG. 5D** shows an example configuration that allows users to roam securely across different networks both inside and outside of the corporate firewall. The mobility server sits behind the firewall. When the mobility client is inside the corporate firewall, connected to the wireless LAN (WLAN), and has been authenticated to the mobility server, packets flow normally and the communication channel between the mobile device and the mobility server (mobile VPN) is protected using IPSec. In this example, the Public Key Infrastructure, passwords and/or any other desired mechanism can be used to perform the key exchange for the IPSec tunnel. For added protection, WLAN access points inside the firewall can be configured to filter all protocols except for a desired one (e.g., IPSec). The mobility server acts as a VPN protecting the data as it traverses the wireless network with IPSec encryption. In this exemplary configuration, the mobility server also acts as a firewall by preventing intruders from accessing the private network. When the mobile device (client) moves into range of the corporate network on a different subnet, it acquires a new point-of-presence (POP) address on the new subnet, negotiates a new secure channel back to the mobility server using IPSec, re-authenticates with the mobility server, and resumes the previously suspended application sessions—all without user intervention being required. The applications can continue to run and the TCP or other connections can be maintained during this network transition since the network transition is transparent to the applications and the mobility server proxies communications on behalf of the mobile device during times when it is unreachable.

[0115] The **FIG. 5E** illustrative network configuration extends the protection of an enterprise firewall to its mobile clients. In this illustrative scenario, the mobility client is configured to use a conventional L2TP/IPSec tunnel to the firewall. IPSec filters on the mobile client can be configured to pass only authenticated IPSec packets to the mobile client's transport protocol stack and reject all other packets. The corporate firewall can be configured to reject all packets except for authenticated IPSec packets for trusted clients; any control channels necessary to set up secure connections; and responses to packets that originate from within the firewall for specifically permitted Internet or other network services. The mobility server located behind the firewall acts as a transport-level, proxy firewall. By proxying all network traffic, user transactions are forced through controlled software that protects the user's device from a wide variety of attacks including for example those using malformed packets, buffer overflows, fragmentation errors, and port scanning. Because the mobility server acts as a transport-level proxy, it can provide this protection transparently for a wide range of applications. Attacks against the network can be blocked by filter rules configured on the firewall and/or the proxy firewall capabilities of the mobility server. Attacks against the mobile device are prevented by the IPSec filter rules configured on the mobile client. Attempts to crack user passwords using sniffer attacks are thwarted by the secure tunnel provided by IPSec.

[0116] **FIG. 5F** shows an additional exemplary illustrative e-commerce model. Like WAP, the **FIG. 5F** arrangement provides optimizations that enhance performance and reliability on slow and unreliable wireless networks. Unlike WAP, the **FIG. 5F** system doesn't allow data to sit on an intermediate server in an unencrypted state. The **FIG. 5F** architecture allows standard web protocols such as HTTP and TLS to be used for e-commerce or other transactions (the web traffic is treated as a payload). The encrypted data is forwarded to its final destination (e.g., the web server) where is can be processed in the same way it would be if two wired peers were performing the same transaction. In addition to optimizations for wireless networks, the **FIG. 5F** system provides seamless roaming between different networks and application session persistence while devices are suspended or out of range of a wireless base station. When combined with the illustrative system's support for public key infrastructure and/or other security mechanisms, those capabilities form a powerful mobile e-commerce platform.

[0117] The scenarios described above are only illustrative—any number of other intermittent, mobile, nomadic or other connectivity scenarios could also be provided.

[0118] Exemplary Integration With IPSec Standards-Based Security Framework

[0119] Generally, the IPSec process of protecting frames can be broadly handled by three logically distinct functions. They are:

[0120] Policy configuration and administration

[0121] Security negotiation/key management

[0122] Privacy processing

[0123] Although these processes are logically distinct, the responsibility for implementing the functionality may be shared by one or more modules or distributed in any manner within an operating or other system. For instance, in the exemplary illustrative client operating system embodiment, the implementation is broken into 3 functional areas or logical modules:

[0124] 1. A Policy Agent module

[0125] 2. A security negotiation and key management (e.g., ISAKMP/IKE) module

[0126] 3. A privacy (e.g., IPSec) module.

[0127] In this illustrative example, the Policy Agent is responsible for the configuration and storage of the configured policy—however it is the IPSec module that actually acts upon the requested policy of the Policy Agent. The preferred exemplary illustrative system provides two different related but separated aspects:

[0128] the first aspect handles IPSec from the mobility client to the firewall or the mobility server; and

[0129] the other aspect handles communication on virtual addresses between the mobility server and peer hosts.

[0130] We first discuss exemplary illustrative communication to and from the mobility client.

Example Mobility Client Architecture

[0131] As part of the preferred embodiment's overall design, network roaming activity is normally hidden from the applications running on the mobility client—and thus, the application generally does not get informed of (or even need to know about) the details concerning mobility roaming. Briefly, as described in the various copending commonly-assigned patent applications and publications referenced above, each of the mobile devices executes a mobility management software client that supplies the mobile device with the intelligence to intercept network activity and relay it (e.g., via a mobile RPC or other protocol) to mobility management server. In the preferred embodiment, the mobility management client generally works transparently with operating system features present on the mobile device to keep client-site application sessions active when contact is lost with the network. A new, mobile interceptor/redirector component is inserted at the conventional transport protocol interface of the mobile device software architecture. While mobile interceptor/redirector could operate at a different level than the transport interface, there are advantages in having the mobile interceptor/redirector operate above the transport layer itself. This mobile interceptor or redirector transparently intercepts certain calls at this interface and routes them (e.g., via RPC and Internet Mobility Protocols and the standard transport protocols) to the mobility management server over the data communications network. The mobile interceptor/redirector thus can, for example, intercept network activity and relay it to server. The interceptor/redirector works transparently with operating system features to allow application sessions to remain active when the mobile device loses contact with the network.

[0132] This arrangement provides an advantageous degree of transparency to the application, to the network and to other network sources/destinations. However, we have found that IPSec is a special case. Between the mobility client and the mobility server or the mobility client and a firewall, IPSec is protecting the packets using the point-of-presence (POP) address. Therefore, in one exemplary embodiment, to allow the existing IPSec infrastructure to operate normally, it should preferably remain informed of the current state of the network. We have therefore modified our previous design to inform IPSec of the change of network status (e.g., so it can negotiate a IPSec session when network connectivity is reestablished) while continuing to shield the networked application and the rest of the operating system from the temporary loss of a network access. Before describing how that is done in one illustrative embodiment, we first explain—for purposes of illustration only the conventional Microsoft Windows 2000/XP operating system IPSec architecture shown in **FIG. 6**. Note that Windows 2000/XP and IPSec is described only for purposes of illustration—other operating systems and security arrangements could be used instead.

[0133] In Windows 2000/XP, the IPSec module is responsible for filtering and protecting frames. For additional information, see for example Weber, Chris, "Using IPSec in Windows 2000 and XP" (Security Focus 12/5/01). Briefly, however, by way of non-limiting illustrative example, before allowing a frame to be processed by the protocol stack or before transmitting the frame out on the network, the network stack first allows the IPSec module a chance to process the frame. The IPSec module applies whatever polices to the frame the Policy Agent requests for the corresponding network identity. In the event that the Policy Agent requires the IPSec module to protect a frame but it does not yet have the required security association (SA) with the peer in accordance with the requested policy, it issues a request to the security negotiation/key management module—in this illustrative case the ISAKMP/IKE (Internet Security Association and Key Management Protocol/Internet Key Exchange) module—to establish one. It is the responsibility of the ISAKMP/IKE module in this illustrative system to negotiate the requested security association and alert the IPSec (privacy) module as to the progress/status of the security association. Once the security association has been successfully established, the IPSec module continues its processing of the original frame.

[0134] In the illustrative embodiment, the Policy Agent uses conventional Microsoft Winsock API's (application programming interfaces) to monitor the state of the network and adjust its policies accordingly. However this is implementation-dependent as other interfaces may also be used to alert this logical component of the network state in other environments. Accordingly, the ISAKMP/IKE module also uses conventional Microsoft Winsock API's to perform security association negotiation as well as track network state changes in one exemplary embodiment.

[0135] Briefly, the above techniques establish a secure IPSec session that is generally tied to a particular IP address and/or port and must be essentially continuous in order to be maintained, as is well known. If the secure session is temporarily interrupted (e.g., because of a lost or suspended connection or a roam) and/or if the IP address and/or port changes, IPSec will terminate it. Unless something is done, terminating the secure IPSec session will cause the mobile application to lose communication even if the network session continues to appear to remain in place. The preferred illustrative exemplary embodiment solves this problem by introducing functionality ensuring that IPSec is passed sufficient information to allow it to react to-the secure session being lost while continuing to shield this fact from the application—and by allowing IPSec to (re)negotiate a secure session once the network connectivity is reestablished using the same or different IP address or port number—all transparently to the networked application. In this way, the exemplary illustrative application is not adversely affected by termination of a previous security session and the establishment of a new one—just as the application is not adversely affected by access to the previous network being terminated and then reestablished (or in the case of roaming, to a new network with a new network identity being provisioned in its place). Meanwhile, the mobility server during such interruptions continues to proxy communications with the peer(s) the mobile device is communicating with so that network application sessions are maintained and can pick up where they left off before the interruption occurred.

[0136] Mobility client-side and server-side support each have different requirements. Therefore the architectures are different in the exemplary illustrative embodiment. The block diagram of an exemplary client architecture is show in **FIGS. 5G and 7**. Note that as compared to conventional **FIG. 6**, we have added two additional components:

[0137] a Policy Agent Hooking component (nmplagnt), and

[0138] a network virtualizing component (nmdrv).

[0139] Briefly, in the preferred illustrative embodiment, the network-virtualizing component virtualizes the underlying client module network while selectively allowing the core operating system's IPSec infrastructure to continue to be informed about network state changes. In the illustrative embodiment, the Policy Agent Hooking component "hooks" certain Policy Agent functions and redirects such processing to the network-virtualizing component so that the normal function of IPSec can be somewhat modified.

[0140] In more detail, in the exemplary embodiment, the network-virtualizing component (nmdrv) uses the services of the existing networking stack and is the layer responsible for virtualizing the underlying client module network. It also initiates and maintains the connection with the mobility server. When a client network application asks for the list of local IP addresses, the network-virtualizing component (nmdrv) intercepts the request and returns at least one of a possible plurality of the mobility client's virtual network identities (e.g. virtual IP addresses).

[0141] However, to continue to allow the inherent IPSec components to operate in a normal fashion, the client architecture should preferably allow the associated IPSec modules to see and track the current point of presence (POP) network address(es). Therefore, in the exemplary embodiment, if a request for the list of network addresses is issued and the request originated in the IPSec process, the network-virtualizing module passes the request along to an inherent network stack without any filtering or modification. Therefore, both the Policy Agent (e.g., polagent.dll in Windows 2000, ipsecsvc.dll in Windows XP) and the ISAKMP/IKE module are kept abreast of the mobility client's current POP address(es).

[0142] In the exemplary embodiment, the network-virtualizing module also tracks address changes. Without this component, the network stack would normally inform any associated applications of address list changes through the conventional application-programming interface, possibly by terminating the application communications end point. In the Microsoft operating systems, for example, this responsibility is normally funneled through the conventional Winsock module, which in turn would then inform any interested network applications of the respective changes. In the exemplary embodiment, the Policy Agent registers interest with Winsock (e.g., using the SIO_ADDRESS_LIST_CHANGE IOCTL via the conventional WSAIoctl function) and waits for the associated completion of the request. The Policy Agent may also be event driven and receive asynchronous notification of such network state changes. Again, in the illustrative exemplary embodiment, the Policy Agent also registers with Winsock a notification event for signaling (e.g., on FD_ADDRESS_LIST_CHANGE via the WSAEventSelect function). When the Policy Agent is alerted to an address list change, it retrieves the current list of addresses, adjusts its policies accordingly and updates the associated policy administration logic. It further informs the Security Negotiation/Key Exchange module, in this case the ISAKMP/IKE module, of the associated state change. The security negotiation/key exchange module (ISAKMP/IKE) module, in turn, updates its list of open connection endpoints for subsequent secure association (SA) negotiations.

[0143] In the exemplary embodiment, Winsock and associated applications are normally not allowed to see address list changes since this may disrupt normal application behavior and is handled by the network-virtualizing component. Therefore, in the preferred exemplary embodiment, another mechanism is used to inform the Policy Agent of changes with respect to the underlying network. To fulfill this requirement in the illustrative embodiment, the services of the Policy Agent Hooking module (nmplagnt) are employed.

[0144] To achieve the redirection of services, the illustrative embodiment employs the facilities of a hooking module (nmplagnt), and inserts the code into the policy administration, security negotiation, and key management (Policy Agent/ISAKMP/IKE) process(es) that are provided as part of the core operating system. In this illustrative embodiment, hooking only certain functions of the Policy Agent module to this redirected code is accomplished via a combination of manipulating the Import Address Table (IAT) together with the use of a technique known as code injection. Injection of the redirected functions is accomplished with the help of conventional operating system APIs (e.g. OpenProcess, VirtualAllocEx, ReadProcessMemory, WriteProcessMemory, and CreateRemoteThread) in the exemplary embodiment. In the preferred exemplary embodiment, once nmplagnt.dll is injected in lsass.exe executable module, it hooks LoadLibrary and FreeLibrary entries in lsasrv.dll so it can detect when the policy agent is loaded and unloaded. Of course, other implementations are possible depending on the particular operating environment.

[0145] Furthermore, the hooking technique in the illustrative embodiment takes advantage of the way in which the Microsoft Windows itself performs dynamic run-time linking. Generally, to facilitate code reuse, Microsoft Windows supports and uses extensively, Dynamic Link Libraries (DLLs). Through the use of DLL technology, a process is able to link to code at run-time. To call a function in a dynamically linked library, the caller must know the location (address) of the specific function in the DLL. It is the operating systems responsibility to resolve the linkage between the code modules and is accomplished via an exchange of formatted tables present in both the caller and callee's run-time code modules. The dynamic library being called contains an Export Address Table (EAT). The Export Address Table contains the information necessary to find the specifically requested function(s) in the dynamic library. The module requesting the service has both an Import Lookup Table (ILT) and an Import Address Table (IAT). The Import Lookup Table contains information about which dynamic library are needed and which functions in each library are used. When the requesting module is loaded into memory for whatever reason, the core operating system scans the associated Import Lookup Table for any dynamic libraries the module depends on and loads those DLLs into memory. Once the specified modules are loaded, the requesting modules Import Address Table is updated by the operating system with the address(location) of each function that maybe accessed in each of the dynamically loaded libraries. Once again, in other environments, different implementations are possible.

[0146] In the exemplary embodiment, after the nmplagnt module is loaded by the prescribed method above, it hooks the Policy Agent's calls to the conventional Microsoft Windows Winsock functions WSASocket, WSAIoctl, WSAEventSelect, closesocket, and WSACleanup. After this process is executed, whenever the Policy Agent module attempts to register for notification of address changes, the request is redirected to the network-virtualizing component. As previously mentioned, the network-virtualizing component by design is aware of changes in network attachment. When it detects a change to the point of presence address, it sends the appropriate notifications to the Policy Agent module. In the illustrative embodiment, this causes the Policy Agent module to query for the current address list. Thus, the Policy Agent and consequently the ISAKMP/IKE module are informed of any address list changes.

[0147] **FIG. 8** is an example of how in the illustrative embodiment a single function from a single DLL might be linked into a calling process. In more detail, the operating system searched the ILT, found a need for target.dll, loaded target.dll into app.exe's address space, located TargetFunc in target.dll's EAT, and fixed up app.exe's IAT entry to point to TargetFunc in target.dll. Now when app.exe calls its stubbed TargetFunc, the stub function will call through the IAT to the imported TargetFunc. Because all of the calls of interest go through the IAT, the preferred exemplary embodiment is able to hook its target functions simply by replacing the corresponding entry for each function in the IAT as shown in **FIG. 9**. This also has the advantage of localizing the hooking. Only the calls made by the requesting module in the target process are hooked. The rest of the system continues to function normally.

[0148] In summary, the illustrative embodiment in one exemplary detailed implementation performs the following steps:

[0149] 1. Call OpenProcess to obtain access to the policy administration, security negotiation, and key management (Policy Agent/ISAKMP/IKE) process(es) and address space

[0150] 2. Use ReadProcessMemory function to find the LoadLibrary function in the associated process(es)' address space.

[0151] 3. Using the VirtualAllocEx function, allocate enough memory to hold the inject illustrative code shown in step 4 in the specified process(es)' address space.

[0152] 4. Use WriteProcessMemory to inject the following code into the memory allocated in step 3:

[0153] LoadLibrary(&targetlibraryname);

[0154] label targetlibraryname:

[0155] "C:\\Program Files\\NetMotion client\\nmplagnt.dll"

[0156] The address of the LoadLibrary function was determined in step 2. The data bytes at label targetlibraryname will vary depending on the name of the module being loaded, where the corresponding module is located, and the operating system environment.

[0157] 5. Call the CreateRemoteThread function to run the injected code.

[0158] 6. Wait for the remote thread to exit.

[0159] 7. Free the allocated memory

[0160] 8. Close the process.

[0161] At the end of these steps, the nmplagnt module has been injected into the policy administration, security negotiation, and key management (Policy Agent/ISAKMP/IKE) process(es) where it is able to redirect the processing of the needed function calls. It is understood that the above code procedure is operating system and processor dependent and is only shown for illustrative purposes, thus not limiting to this specific sequence or operation. Furthermore, the executable code responsible for adding these components to the operating environment can be provided to the mobile device via storage on a storage medium (e.g., optical disk) and/or by downloading over the network

[0162] A similar method is employed using the FreeLibrary function instead of LoadLibrary function to reverse the hooking process and to unload the nmplagnt module. For the sake of brevity, the description is kept minimal, as anyone schooled in the art should be able to achieve the desired results.

Example Mobility Server Architecture

[0163] Using IPSec methodology for communication between the mobility server and peer hosts is a different set of problems to solve—although it uses some of the same techniques used on the mobility client. **FIG. 10** shows an exemplary server architecture. In the exemplary illustrative embodiment, the mobility server MS can also be based on a Windows 2000/XP (or any other) operating system. In this particular illustrative implementation, a hooking module is also used in the illustrative embodiment—but the functions intercepted by the hooking module in the case of mobility server MS are redirected to the proxy and filter modules that are also supplied by the preferred exemplary embodiment, instead of the network-virtualization module.

[0164] In the exemplary mobility server, a proxy driver (nmproxy) can be used to implement the bulk of the mobility server functionality. However, in one exemplary implementation, there are three separate problems to solve for which three additional logical modules are used. They are:

[0165] a network identity mapping driver (nmprtmap),

[0166] an IPSec filter driver (nmipsec), and

[0167] the security negotiation hooking library (nmike).

[0168] The first problem is how to manage virtual addresses for the mobility clients. Although it is possible in some network stack implementations to assign multiple addresses to the inherent networking stack components of the operating system, some systems do not support such functionality. To support the more restrictive implementation, the illustrative example embodiment employs the use of an identity mapping technique. It will be appreciated that the techniques herein are both compatible with and complimentary to either implementation, and such identity mapping functionality allows the security functionality to successfully operate within the more restrictive environments. The illustrative mobility server opens a communications endpoint associated with a local address and port and then identity maps between the corresponding virtual address(es) and port(s) before packets are processed by the protocol stack during reception and before they are transmitted out on the network. That mapping is the job of the network identity mapping module (nmprtmap) in one exemplary embodiment. For example, assume an application on a mobility client opens TCP port 21 on virtual address 10.1.1.2. Through the use of previously-defined mechanisms (see for example U.S. patent application Ser. No. 09/330,310 filed Jun. 11, 1999, entitled "Method And Apparatus For Providing Mobile and Other Intermittent Connectivity In A Computing Environment), this request is transferred from the mobility client to the mobility server. In response, the MS opens the connection on its local address 10.1.1.1 on port 2042 and registers the appropriate mapping with the network identity mapping module. When the mobile client together with proxy driver (nmproxy) then wishes to send data on its newly opened connection, the packet generated by the inherent networking stack will have a source address of 10.1.1.1 port 2042. The network identity mapping module will then match the frame's protocol/address/port tuple against its mapping table and replace the source address with 10.1.1.2 port 21 before the packet is transmitted on network. The reverse operation is performed for received packets. Using this network identity mapping technique allows the mobility server to communicate to peer systems using virtualized addresses without requiring modification to the core operating system transport protocol stack

[0169] The second problem is a direct result of this mapping technique. Because the network identity mapping module logically operates below IPSec module (i.e. processes frames before during reception and after during transmission), it cannot directly manipulate IPSec protected frames without corrupting the packets or being intimately involved in the privacy or authenticating process. To address this issue, in one exemplary embodiment, the aforementioned IPSec filter module (nmipsec) inserts itself between the operating systems networking stack components and the associated IPSec modules. The filter module inspects each outgoing packet before IPSec protects the packet and each incoming packet after IPSec removes any encoding. Once in control of the frame, it consults the network identity mapping module(nmprtmap) to determine whether or not the frames source or destination identity should be mapped. In this way, the functionality of the mapping logic is moved to a level where is can perform its function without interfering with the IPSec processing.

[0170] Hooking the link between the IPSec and networking stack components is implementation and operating system dependent. In the illustrative exemplary embodiment, again the hooking process is completed by the manipulation of tables that are exchanged between the inherent IPSec and networking stack modules—but other implementations and environments could rely on other techniques. In the illustrative embodiment the IPSec filter module (nmipsec) loads before the IPSec module but after the transportprotocol module. When the IPSec module attempts to exchange its function table with the transport protocol components, the IPSec filter module (nmipsec) records and replaces the original function pointers with it's own entry points. Once the associated tables are exchanged in this manner, the IPSec filter module (nmipsec) can manipulate the contents of and control which packets the inherent IPSec module operates on.

[0171] The third issue is where the hooking techniques also used by the mobility clients is employed. As mentioned previously, due to the mapping technique employed in one exemplary implementation, the inherent networking stack has no knowledge of the mobility client's virtual address(es). Consequently, the policy administration, security negotiation, and key management (Policy Agent/ISAKMP/IKE module) process(es) are also not cognizant of these additional known network addresses. Therefore, there are no IPSec security policies to cover frames received for or transmitted from the mobility client's virtual address(es). Furthermore the security negotiation module (in this case the ISAKMP/IKE module) has no communications end point opened for which to negotiate security associations for the mobility client. To address this issue in the exemplary embodiment, the security negotiation hooking module(nmike) can employ the same hooking methodology described for the mobility client and illustrated in **FIG. 8**. The security negotiation hooking module (nmike) intercepts any address change notification request. When the proxy modules registers or deregisters a mobility client's virtual address(es) with the network identity mapping module(nmprtmap), it also informs the security negotiation hooking module(nmike). This module in turn then informs the policy administration module (Policy Agent) of the respective change. When either the policy administration (Policy Agent) or the security negotiation (ISAKMP/IKE) module requests a list of the current addresses via the conventional Microsoft Windows GetIpAddrTable function call, the security negotiation hooking module (nmike) intercepts the request and adds all of the current virtual addresses to the returned list. When the policy administration module (Policy Agent) sees the respective virtual addresses in the list, it treats them as actual addresses and creates the appropriate policies for the IPSec module. In response to the modification of the network address list, the security negotiation (ISAKMP/IKE) module will attempt to open and associate a communications endpoint for each address in the list. However, as mentioned previously, since the inherent networking stack in the illustrative embodiment is ignorant to the fact of these additional network addresses due to the aforementioned mapping methodology, this operation will generally fail. To solve this problem, the security negotiation hooking module (nmike) intercepts the request to the conventional Microsoft windows Winsock bind function and modifies the requested virtual address and port with a INADDR_ANY. Once the endpoint is bound through the inherent transport protocol stack, the security negotiation hooking module (nmike) employs the services of the network identity mapping module (nmprtmap) and creates a mapping between the actual address and port associated with the newly established communications end point to the virtual address and the assigned port for security negotiations (in this case port 500 is the standard ISAKMP port). Finally, the security negotiation hooking module (nmike) registers the actual address and port with IPSec filtering module (nmipsec) to instruct the module to pass packets to and from the specified address without further IPSec filter processing.

[0172] All documents referenced herein are incorporated by reference as if expressly set forth herein.

[0173] While the invention has been described in connection with practical and preferred embodiments, it is not to be so limited. Specifically, for example, the invention is not limited to IPSec or Microsoft operating systems. IPSec and related technologies can be arranged in a number of manners, executing with some of the required algorithms executing either in software or hardware. To wit, certain implementations may include hardware accelerator technology for the ciphering process, etc. Many network interface and computer manufactures have commercially available products that are used for this exact purpose. It is to be appreciated that the above specifications however describes the logical placement of required functionally and may actually execute in a distributed fashion. Accordingly, the invention is intended to cover all modifications and equivalent arrangements within the scope of the claims.

We claim:

1. A method of maintaining network communications with a mobile or other intermittently connected computing device executing at least one networked application that participates in at least one network application session, comprising:

(a) detecting the occurrence of an event affecting network communications with the computing device, and

(b) in response to said detection, terminating, instantiating, and/or reinstantiating an IP Security session for use by said computing device while maintaining said network application session(s).

2. The method of claim 1 wherein said detecting comprises detecting a change in network point of attachment.

3. The method of claim 1 wherein said detecting comprises detecting that an interruption of network connectivity has caused a previous IP Security session to be terminated.

4. The method of claim 1 wherein said detecting comprises detecting that the mobile device's network identity has changed.

5. The method of claim 1 wherein said detecting comprises detecting that the mobile device has roamed to a different network or subnetwork.

6. The method of claim 1 wherein said step (b) comprises negotiating a new IP Security session to replace a previous, lost IP Security session in a manner that is transparent to the networked application.

7. The method of claim 1 wherein said step (b) includes using IPSec to create a secure tunnel through the network.

8. The method of claim 1 further including applying policy rules to selectively allow, deny or delay the flow of network communications over said IP Security session.

9. The method of claim 1 further including centrally managing and distributing policy regarding the establishment of said IP Security session from a central authority.

10. The method of claim 1 further including securely proxying said mobile device communications.

11. The method of claim 1 further including terminating a previous IP Security session based on said detecting.

12. A method of modifying an operating environment having at least one software component, said operating environment using transport engine protocols and running at least one application, the method comprising:

(a) transparently and selectively injecting computer instructions into said operating environment; and

(b) redirecting the execution path of said at least one software component to achieve additional functionality while maintaining binary compatibility with said operating environment component(s), said transport engine protocols and said applications.

13. The method of claim 12 wherein said redirecting is performed based on process name.

14. A method of providing data communications in a mobile computing environment, said environment including at least one device using at least one network interface for network applications and operating system components, comprising:

(a) selectively and transparently virtualizing said at least one network interface, thereby shielding said network applications and operating system components from at least some characteristics of said mobile computing environment, and

(b) allowing other said components to remain cognizant of at least interruptions in connectivity and changes in network point of attachment.

15. A method for providing data communications in an environment including at least one device using at least one network interface for network applications and operating system components, comprising:

(a) selectively virtualizing said at least one network interface, thereby shielding said network applications and operating system components from at least some adverse events that may otherwise disturb communications; and

(b) using said virtualized network interface to conduct data communications.

16. The method of claim 15 wherein said adverse events include changes in network point of attachment.

17. The method of claim 15 wherein said adverse events include periods of network disconnectedness.

18. A method for using plural IP Security sessions over a plurality of network interfaces associated with at least one network point of attachment, comprising:

(a) distributing network application communications to simultaneously flow over said plural IP Security sessions, and

(b) multiplexing/demultiplexing said distributed communication flows into corresponding higher layer communications sessions.

19. The method of claim 18 further including applying policy rules to selectively allow, deny, or delay the flow of network communications over at least one of said plural IP Security sessions.

20. The method of claim 18 further including centrally managing and distributing policy regarding the establishment of said plural IP Security sessions from a central authority.

21. A method comprising:

(a) facilitating the creation of plural IP Security sessions; and

(b) selectively allowing, denying and/or delaying the flow of network communications over at least one of said plural IP Security sessions based at least in part on applying policy rules.

22. The method of claim 21 further including centrally managing and distributing said policy rules from a central authority.

23. A method of administering secure network connections comprising:

(a) establishing IP Security sessions within a computing network; and

(b) centrally managing and distributing policy regarding the establishment of said IP Security sessions from a central authority.

24. A method of proxying mobile communications comprising:

(a) establishing communications with a mobile device;

(b) establishing communications with an ultimate peer of said mobile device; and

(b) instantiating at least one of a possible plurality of IP Security sessions with said ultimate peer on behalf of said mobile device.

25. The method of claim 24 wherein said mobile device includes a client and (a) comprises establishing client-server communications.

26. A method of proxying mobile communications comprising:

(a) establishing at least one IP Security session between said mobile device and a communication peer thereof; and

(b) maintaining said IP Security session with said communication peer during periods when said mobile device is unreachable.

27. A method of managing IP Security sessions between a mobility server and an ultimate communications peer, comprising:

(a) establishing at least one IP Security session between said mobility server and said ultimate communications peer; and

(b) automatically terminating said IP Security session in response to occurrence of a predetermined event.

28. The method of claim 27 wherein the predetermined event is selected from the group comprising link activity, application session inactivity, and termination of a communications end point.

29. A method of providing secure communications between a mobility client having a network identity, a mobility server and an ultimate communications peer, comprising:

(a) establishing at least one IP Security session between the mobility server and the ultimate peer; and

(b) securely maintaining said IP Security session even when the network identity of said mobility client changes.

30. In a system for maintaining network communications with a mobile or other intermittently connected computing device executing at least one networked application that participates in at least one network application session, said system comprising:

a detector that detects the occurrence of an event affecting network communications with the computing device, and

a security module that, in response to said detection, instantiates or reinstantiates an IP Security session for use by said computing device while maintaining said network application session(s).

31. The system of claim 30 wherein said detector detects a change in network point of attachment.

32. The system of claim 30 wherein said detector detects that an interruption of network connectivity has caused a previous IP Security session to be terminated.

33. The system of claim 30 wherein said detector detects that the mobile device's network identity has changed.

34. The system of claim 30 wherein said detector detects that the mobile device has roamed to a different network or subnetwork.

35. The system of claim 30 wherein said security module negotiates a new IP Security session to replace a previous, lost IP Security session in a manner that is transparent to the networked application.

36. The system of claim 30 wherein said security module uses IPSec to create a secure session through the network communication.

37. The system of claim 30 further including a policy manager that applies policy rules to selectively allow, deny or delay the flow of network communications over said IP Security session.

38. The system of claim 30 further including a central policy management authority that centrally manages and distributes policy regarding the establishment of said IP Security session.

39. The system of claim 30 further including a mobility server that securely proxies said mobile device communications.

40. The system of claim 30 wherein the security module terminates a previous IP Security session based on said detection.

41. An operating environment having at least one software component, said operating environment using transport engine protocols and running at least one application, the environment further comprising computer instructions transparently and selectively injected therein, wherein the injected computer instructions include a redirector that redirects the execution path of said at least one software component to achieve additional functionality while maintaining binary compatibility with said operating environment component(s), said transport engine protocols and said applications.

42. The environment of claim 41 wherein said redirector redirects said execution path based on process name.

43. A mobile computing environment including at least one device using at least one network interface for network applications and operating system components, said environment comprising:

(a) instructions that selectively and transparently virtualize said at least one network interface, thereby shielding said network applications and operating system components from at least some characteristics of said mobile computing environment, and

(b) further instructions that allow other said components to remain cognizant of at least interruptions in connectivity and changes in network point of attachment.

44. An environment including at least one device using at least one network interface for network applications and operating system components, said environment comprising:

instructions that selectively virtualize said at least one network interface, thereby shielding said network applications and operating system components from at least some adverse events that may otherwise disturb communications; and

additional structure that uses said virtualized network interface to conduct data communications.

45. The environment of claim 44 wherein said adverse events include network point of attachment.

46. The environment of claim 44 wherein said adverse events include periods of network disconnectedness.

47. A system for using plural IP Security sessions over a plurality of network interfaces associated with at least one network point of attachment, comprising:

a data distributor that distributes network application communications to simultaneously flow over said plural IP Security sessions, and

(b) a multiplexer/demultiplexer that multiplexes and demultiplexes said distributed communication flows into corresponding higher layer communications sessions.

48. The system of claim 18 further including applying policy rules to selectively allow, deny, or delay the flow of network communications over at least one of said plural IP Security sessions.

49. The system of claim 47 further including a central authority that centrally manages and distributes policy regarding the establishment of said plural IP Security sessions.

50. A system comprising:

(a) a security framework that facilitates the creation of plural IP Security sessions; and

(b) a policy agent that selectively allows, denies and/or delays the flow of network communications over at least one of said plural IP Security sessions based at least in part on policy rules.

51. The system of claim 50 further including a central authority that centrally manages and distributes said policy rules.

52. A system for administering secure network connections comprising:

a security framework that establishes IP Security sessions within a computing network; and

a central authority that centrally manages and distributes policy regarding the establishment of said IP Security sessions.

53. A mobility proxy comprising:

a communications structure that establishes communications with a mobile device and with an ultimate peer of said mobile device; and

a security component that instantiates at least one of a possible plurality of IP Security sessions with said ultimate peer on behalf of said mobile device.

54. The system of claim 53 wherein said mobile device includes a client and mobility proxy comprises a server.

55. A system for proxying mobile communications comprising:

communications means for establishing at least one IP Security session with said mobile device and a communication peer thereof; and a means for maintaining said IP Security session with said communication peer during periods when said mobile device is unreachable.

56. A system for managing IP Security sessions between a mobility server and an ultimate communications peer, comprising:

means for establishing at least one IP Security session between said mobility server and said ultimate communications peer; and

means for automatically terminating said IP Security session in response to occurrence of a predetermined event.

57. The system of claim 56 wherein the predetermined event is selected from the group comprising link activity, application session inactivity, and termination of a communications end point.

58. A system for providing secure communications between a mobility client having a network identity, a mobility server and an ultimate communications peer, comprising:

means for establishing at least one IP Security session between the mobility server and the ultimate peer and the mobility client and the mobility server; and

means for securely maintaining said IP Security session even when the network identity of said mobility client changes.

59. A storage medium storing:

a first set of instructions that inserts a policy agent hooking runtime linkable module into an operating system having a policy agent and an IPSec infrastructure, said hooking module informing the policy agent of network state changes; and

a second set of instructions that inserts a network interface virtualizing driver into said operating system, said virtualizing driver virtualizing a client module network and initiating mobility server connections while selectively allowing the IPSec infrastructure to continue to be informed about network state changes.

60. A method of preparing a mobile device for secure communications, said mobile device having an operating environment including a policy agent and an IPSec infrastructure, said method comprising:

downloading over a computer network onto the mobile device and executing with the mobile device, a first set of instructions that insert a policy agent hooking runtime linkable module into the operating environment, said hooking module informing the policy agent of network state changes; and

downloading over the computer network and executing with the mobile device a second set of instructions that inserts a network interface virtualizing driver into said operating environment, said virtualizing driver virtualizing a client module network and initiating mobility server connections while selectively allowing the IPSec infrastructure to continue to be informed about network state changes.

* * * * *